2

COMPUTER SYSTEMS LABORATORY

STANFORD UNIVERSITY · STANFORD, CA 94305-4055

AD-A228 744

Microsupercomputers: Design and Implementation

Stanford University
Computer Systems Laboratory

Semi-Annual Technical Report

Defense Advanced Research Projects Agency

For the period of April 1990 - October 1990

Contract Number: N00014-87-K-0828

Principal Investigator
John L. Hennessy

Associate Investigator
Mark A. Horowitz

Semi-Annual Technical Progress

April 1990 - October 1990

Contract No. N00014-87-K-0828

Order No. 1133

R & T Project Code: 4331685

Accesion For	
NTIS CRA&I	✓
DTIC TAB	☐
Unannounced	☐
Justification	
By Per call	
Distribution /	
Availability Codes	
Dist	Avail and / or Special
A-1	

Dist. "A" per telecon Dr. Andre van Tilborg. Office of Naval Research/ code 1133.
VHG 11/13/90

This work is supported by the Defense Advanced Research Projects Agency and the Office of Naval Research.

Table of Contents

1. Project Goals:

1. Investigate fundamental properties of parallel programs and the implications for multiprocessor architectures and parallel programming and compilers;

2. Explore architectural approaches that can be used to build scalable multiprocessors capable of supporting a shared memory programming paradigm,

3. Develop compiler and programming language technology that supports the construction of efficient, machine-independent parallel programs;

4. Investigate architectural approaches that will lead to substantial improvements in uniprocessor performance and that can be incorporated in scalable multiprocessors,

5. Develop CAD tools and VLSI technology needed to construct high performance machines. [illegible]

[illegible]

2. Executive Summary

DASH Hardware: The 16 processor prototype of DASH is now very close to completion. All the hardware is either in fabrication or being debugging. We are shooting for an operational prototype this Fall.

DASH Operating System: The operating system modifications have been keeping pace with the hardware. All essential modifications necessary to bring up the operating system on the DASH prototype have been completed. We have already successfully booted the modified OS on a single cluster while using the new locks and interrupts provided by the directory-controller board.

Parallel Simulation and Tracing Facility: We have continued the development and use of our software tracing and simulation system, Tango. We have extended Tango to allow multiple applications to be run at the same time with a user supplied scheduling strategy. We have studied the performance gains of weak memory consistency models over sequential consistency for scalable multiprocessors. Another study has evaluated the benefits of software-controlled prefetching in multiprocessors. For the scientific applications we studied, we found that prefetching was easy to add and it increased the performance by 200-300%.

Compiler Managed Parallelism: We have developed new algorithms for two basic parallelizing compiler modules: the data dependence analyzer and the loop transformer. While current methods sacrifice potential parallelism to improve compiler efficiency, we demonstrate that it is possible to test data dependence exactly and efficiently.

Data Management: We have discovered that the behavior of caches on blocked numerical code is quite unusual and must be accounted for in the design of such algorithms. Especially for direct-mapped caches, the conventional wisdom of trying to use the entire cache, or even a fixed fraction of the cache, is incorrect. The cache performance can be significantly improved by choosing the block size according to the matrix size or by copying data to contiguous memory locations.

Sparse Matrix Factorization: We have been exploring new ways to efficiently solve large sparse systems of equations. Using the concept of supernodes, we have developed techniques that make very effective use of caches. On an 8-processor Silicon Graphics system, SGI 4D-380, we obtain over 40 MFLOPS of sustained performance.

Parallel Languages: To compensate for the weakness of parallelizing compilers in extracting coarse-grain parallelism, we have developed a parallel programming language called Jade. Starting with a sequential program, a programmer simply augments those sections of code to be parallelized with side effect information. The Jade system finds not just static parallelism but also parallelism that can only be derived at run time. Using Jade can significantly reduce the time and effort required to develop a parallel version of an imperative application with serial semantics. We have implemented a prototype Jade system and also several applications using this language.

Protocol Verification: We are applying protocol verification techniques to the cache coherence protocol for DASH. Although this project is in the initial stages, we have completed a reduced description of the protocol which has been sufficient to find some cases that were not covered in the documentation of the protocol. We are continuing to enhance the DASH description and specification, with the goal of either proving it correct or accelerating the debugging of the DASH prototype.

VLSI CAD Tools: We have also continued our work in tool support for VLSI design. Our recent work has been in three areas: incremental simulation, power supply noise analysis, and BiCMOS simulation. In the area of incremental simulation, we have completed and begun distributing IRSIM, an incremental switch-level simulator. This tool is a variant of RSIM, that allows the user to change the netlist after a simulation has been performed and quickly find the effects of the change. We are continuing work on Bisim which started out as an attempt to create an RSIM-like tool for BiCMOS circuits.

We are finishing up our work on Ariel, an analysis tool that allows the designer to calculate the noise on the power supply lines in the integrated circuit. This program first extracts the resistance of the supply lines, then estimates the supply current and then uses this information to find the voltage drops.

Self-Timed Circuits: We worked on a number of VLSI chips this period. In the self-timed area, we designed, fabbed and tested a new self-timed divider, that is able to find a new quotient bit every 3ns. The key point of this design is that the self-timed control does not add any delay to the circuit, the delay is completely set by the datapath. We call this new style Zero-Overhead Self-Timed Design.

High-Speed Memory: We have also been working on improving our designs of high-speed BiCMOS memories. Previously we demonstrated a 64K 4ns SRAM. During this period, we have looked at ways to make the design more robust, to see if a commercial memory would be possible. The results look very promising, and we are now looking at other ways on improving both the speed and noise-margins in the design.

3. Technical Progress

3.1 Parallel Processor Architecture

Our primary focus over the last 6 months has been the construction of the DASH prototype. The prototype is now close to completion and we hope to have working hardware and software within the next few weeks.

Our first subtask was making the modifications necessary to the processor, memory, and I/O boards of the Silicon Graphics (SGI) 4D/340 systems used in DASH. For example, we needed to modify the processor boards and the I/O board (which has the arbiter) to force the processor off the bus when it makes a remote-memory request and eventually be able to respond to the processor when data has been retrieved. The memory boards needed to be modified to recognize non-zero base addresses, plus a host of other changes. All such modifications have now been completed and tested.

Our second subtask was to complete the design of the directory board and to build it. Since our last report, we have split the single directory board into two boards---the directory-controller board and the reply-controller board. In addition to accommodating all the main logic, this allowed us to put a powerful performance monitor on each cluster. The directory-controller board now contains the directory memory and the associated state machines, the performance monitor, and one-half of the network interface. The reply-controller board contains the reply controller (handles replies to this cluster's remote requests), the pseudo-CPU (services requests from remote CPUs), the remote-access cache, and the other half of the network interface. The design of both boards is now complete. In fact, the directory-controller board has already been fabricated. We have inserted the board into a modified SGI system and have tested much of its functionality. The board seems to be working according to the specifications. The reply-controller board was sent for fabrication on September 21 and should be back on October 5. Within a few days from that we will be ready to debug a multiple cluster DASH system.

Before the boards were sent for fabrication, considerable effort was spent on ensuring correctness through simulation. We built a detailed functional simulator of DASH that closely modeled the DASH hardware. Several parallel applications were run on this simulator and millions of clock cycles were simulated. We also ran the DASH protocol verifier, a psuedo-random tester that exercises the coherence protocol, to ensure that a large percentage of possible protocol interactions were tested. About 200K vectors, directly generated from this functional simulator, were used to do gate-level simulations using the VALID CAE software.

DASH is designed to be a general-purpose multiprocessor, and not as an attached machine to some host. As such, it will support a full fledged operating system. In close cooperation with Silicon Graphics, we are modifying the existing operating system on the 4D/340 (IRIX, a variation of UNIX System V) to work on DASH. Because of the hierarchical nature of the DASH multiprocessor, as compared to the flat structure of the 4D/340, several major changes are required to the kernel. At this point, the boot up sequence for the multiple cluster DASH has been defined. Most kernel data structures have been adapted for the DASH architecture, and special tools have been developed in order to automate the process of configuring the kernel for different cluster configurations. The kernel locking system, the user locking system and the user MP library are currently using the DASH hardware locks (with special added code to cover the absence of the RC board). The capability of attaching processes or sets of threads to a given processor has been extended to allow the attachment to clusters. Inter-cpu interrupts are currently being generated through the DASH hardware interrupt generation facility. The operating system is now at a stage where it is ready to be run and debugged on the multiple cluster DASH when it is powered up in early October.

In addition to the work being done on the prototype, we have been doing a number of other architectural studies. One important topic is the performance benefits from weak memory consistency models. Although, a number of models have been proposed in the literature, no detailed performance results have been reported. We have done detailed simulation studies to characterize the benefits [7]. Our results show that the sequentially consistent models have significantly worse performance than the less strict models such as processor consistency and release consistency. For the four benchmark applications studied, the less strict models were shown to improve the processor utilization by as much as 10-50% over a sequentially consistent implementation. The gains are expected to increase with larger memory latencies that will be seen in future machines. We further show that most of the benefits achieved by the less strict models are due to buffering of writes and allowing reads to bypass pending writes. The ability to pipeline writes is not as critical to performance, especially when reasonably deep write buffers were used. The above results were shown for processors with blocking loads. We are currently doing studies for processors with non-blocking loads, where the benefits are expected to increase.

In another study [19], we evaluated the effectiveness of non-binding software-controlled prefetching (as proposed in the Stanford DASH multiprocessor) in helping tolerate large memory latencies observed in scalable multiprocessors. In adding prefetching to applications, we found both the non-binding and software-controlled aspects to be essential to obtaining good performance benefits. Non-binding prefetching allowed us to fetch data far in advance, even though there was a possibility that data may not be used or that it may be modified before use. Software control allowed us to be selective in only prefetching data that was likely to miss in the cache, thus reducing overhead. It also allowed us to move prefetches earlier than would have been possible in hardware schemes that use a limited lookahead window. Our results show that for applications with regular data access patterns---we evaluated a particle-based simulator used in aeronautics and an LU-decomposition application---prefetching can be very effective. It was easy to augment the applications to do prefetching and it increased their performance by 200-300% when we prefetched directly into the processor's cache. However, for applications with complex data usage patterns, for example applications that make extensive use of pointers and linked lists, prefetching was less successful. After much effort, the performance of a distributed-time logic simulation application could be increased only by 30%.

3.2 Modeling of Cache Coherence Directories

We have been studying various aspects of using directories to keep processor caches consistent. These cache coherence directories record the caches that contain copies of data, thereby allowing messages to be sent to only those caches that must receive them, rather than broadcasting to every cache in the system. Directory-based coherence schemes consume substantially less interconnection bandwidth than their broadcast-based counterparts, and give the designer the flexibility of choosing any general point-to-point network to interconnect the caches and main memory.

We have recently developed a model of reference behavior that allows us to predict the performance of limited pointer directories under different program workloads. This will allow us to properly evaluate several important trade-offs in the design and implementation of these directories. For example, since the model predicts the number of pointers needed under different circumstances, we can find the point of diminishing returns where adding additional pointers to each entry yields negligible performance gains. Another application of our model is evaluating the mechanisms used when the available pointers in an entry are exhausted. Developing schemes with good overflow behavior will allow us to cut the number of pointers required even further.

The modeling effort is ongoing and is moving forward rapidly. Several useful models have been developed and verified using simulations driven by address traces of parallel programs. We are now evaluating early results from the models. Though our conclusions are necessarily still preliminary, it appears that limited pointers schemes can substantially reduce the width of each directory entry for machines with hundreds or thousands of processors.

4. Parallel Software

4.1 Compiler Research

The state of our compiler is that it has started to produce correct sequential code. It can now generate correct, unoptimized code for several of the large programs from the Perfect Club benchmarks. These Fortran programs were first converted into C, then our intermediate format, and finally into MIPS assembly code. This shows that the basic implementation is solid. The various advanced modules are now at different stages of implementation; they include data dependence analysis, various scalar optimizations including register allocation, flow graph optimizations for parallelism, the loop transformer, array renaming, superscalar instruction scheduler, and the run-time system. Our goals are not to just develop new compiler technology, but also evaluate them experimentally.

We have developed a new approach to data dependence analysis. Data dependence testing is the basic step in detecting loop level parallelism in Fortran-like languages. Current methods sacrifice potential parallelism to improve compiler efficiency. We have developed a small set of efficient algorithms, each one exact for special case inputs. Combined, they are exact for almost all the cases we see in practice. We also introduce a memorization technique to save results of previous tests, thus avoiding calling the data dependence routines multiple times on the same input. Finally, we introduce new pruning techniques, allowing us to also compute direction vectors more efficiently. We applied the algorithm to the Perfect Club Benchmarks, a set of 13 scientific Fortran programs ranging in size from 500 to 18,000 lines. We show both that our memorization technique allows us to eliminate most tests and that the large majority of remaining tests can be solved exactly using our simple algorithms. Combining these with a moderately expensive backup test, we are able to be exact in every case at a very reasonable cost.

We have developed a theory that unifies many existing loop transformations, including loop interchange or permutation, skewing, reversal, tiling, and combinations of these elementary transformations [34]. This theory provides the foundation for solving an open question in compilation for parallel machines: which loop transformations and, in what order, should they be applied to achieve a particular goal, such as maximizing parallelism or data locality. We have developed efficient algorithms for these problems that will find the optimal solution in most cases.

To maximize locality, the algorithm can be divided into two steps. The first step of loop transformations and blocking can be performed without deciding on the block size of the final code. The block sizes can then be chosen after the loops have been transformed. While it is well known that blocking numerical code is an important optimization to increase the performance of memory hierarchy in general, the behavior of caches on blocked code is not known. The conventional wisdom is to choose the block sizes that will use some fixed fraction of the cache. Our experiment [17] suggests otherwise. We have obtained a significant set of data on the performance of caches for blocked code and evaluated several optimizations for it. The data is obtained by a theoretical model of data conflicts in the cache, which is validated by substantial amounts of simulation. We show that: the performance is extremely sensitive to the stride of data accesses. To minimize the expected number of cache misses, the code should attempt to use only a small percentage of the cache within each block. Increasing the block size to use more of the cache may severely degrade the performance of the machine. Using different block sizes for different matrices can improve the miss rates and reduce the variance in performance for different problem sizes. Finally, whenever possible, it is beneficial to copy non-contiguous data into consecutive locations.

4.2 Solving Sparse Matrix Problems Efficiently

We are currently looking at the problem of efficiently solving large sparse systems of equations on hierarchical memory uni- and multi-processors [20, 21, 22, 23]. This is an important problem; the solution of such systems is the bottleneck in a wide variety of domains, including linear programming, device simulation, and computational fluid dynamics. Hierarchical memory multiprocessors offer the potential to solve such problems both extremely quick and cost-effective.

The particular method we have been investigating for solving sparse systems is Cholesky factorization. The main focus of our research has been on the use of blocking techniques to improve the performance of sparse Cholesky factorization. Blocking has been successfully exploited by a number of researchers to reduce cache miss-rates, and thus improve performance, for dense linear algebra algorithms. The LAPACK dense linear algebra package, for example, makes extensive use of blocking techniques. Blocking can briefly be described as a reordering of the steps of an algorithm to increase data locality. A block of data that fits in a fast level of the memory hierarchy is loaded into this fast memory and reused many times before another block is loaded. The blocking done for dense linear algebra algorithms relies heavily on the extremely regular structure of a dense matrix computation. Sparse problems, in general, lack this regularity and thus are less amenable to blocking.

Blocking is made possible in sparse Cholesky factorization by the presence of large sets of columns in the factor matrix, called supernodes, that have nearly identical non-zero structures. Originally examined in the context of vector supercomputers, these supernodes dramatically increase the regularity of the sparse factorization computation. For vector supercomputers, this increase results in increased vectorization. In the context of hierarchical memory machines, we have used supernodes to make effective blocking possible. The result is substantially higher performance.

A blocked approach to sparse Cholesky factorization has a number of implications for parallel sparse factorization. The primary implication is that it greatly increases the task grain size, making distribution of work among the processors more difficult. Through heuristic partitioning of tasks, however, a reasonable balance can be reached between the efficiency of a fully blocked code and the load balancing problems that would result from too large a task grain size.

The major contributions of our work are as follows. We have performed an in-depth study of the performance of the important sparse Cholesky factorization computation on a class of machines that has so far not been considered in this context. Previous studies of Cholesky factorization have considered its performance on either vector supercomputers or scalar machines without cache memory, both of which present very different considerations. We have also proposed a number of new blocking approaches for reducing cache miss-rates. We have extended these techniques to hierarchical memory multiprocessor systems, where blocking greatly complicates previous approaches to parallel sparse factorization. We have proposed task partitioning techniques that overcome the majority of these complications. The results of our studies have indicated that blocking is an extremely effective technique for improving the performance of sparse Cholesky factorization on hierarchical memory machines. We have obtained computation rates of approximately 40 double-precision MFLOPS for a range of sparse problems on an 8-processor Silicon Graphics 4D/380 multiprocessor, more than twice the performance of unblocked approaches.

4.3 Language Research

Our language research is directed at complementing our compiler research. Although parallelizing compilers have had some success with statically analyzable parallel loops, they have been unable to automatically extract available task-level concurrency. The reasons are that they are unable to automatically partition programs into a reasonable set of coarse-grain tasks, and compilers' conservative dependence analysis generates spurious data dependences, which unnecessarily serialize the computation. Programmers, on the other hand, have the high level program knowledge necessary to exploit coarse-grain concurrency. A programmer usually has little difficulty determining both a reasonable task decomposition for a program and the precise side effect information which enables the identification of concurrently executable tasks; the programmer's main problem is getting the compiler and run-time system to understand and utilize this information. To solve this problem, we have developed a programming language called Jade which enables the exploitation of coarse-grain concurrency by allowing a programmer to easily express this program knowledge [16].

Starting with a sequential program, a programmer simply augments those sections of code to be parallelized with side effect information. The compiler and run-time system use this side effect information to concurrently execute the program while respecting the program's data dependence constraints.

Languages that contain explicit synchronization primitives can be viewed as parallel "assembly" programming languages. Programs written in these languages must be tuned when ported to machines with different characteristics. Jade programs preserve the programmer's high-level program knowledge, making it possible for the compiler and run-time system, not the programmer, to exploit this knowledge when implementing machine-dependent optimizations.

5. Uniprocessor Architecture

5.1 Super-Scalar Computers

For the past six months, we have concentrated effort on the hardware and software aspects of the new architecture that we proposed for superscalar processors. This architecture tries to combine the advantages of dynamic and static scheduling techniques, while minimizing the short-comings of each, to increase the performance of non-numerical applications.

Early in this six-month period, we completed a paper [26] that describes the basics of the new superscalar architecture and gives preliminary results on its effectiveness. We call our new architecture TORCH. This architecture introduces boosting as a technique to allow the compiler to specify speculative execution. Boosting allows instructions to be scheduled for execution before a conditional branch upon which the instructions are dependent is scheduled. When an instruction is scheduled before a controlling conditional branch, the sequential instruction becomes a boosted instruction which is signified by a separate encoding. The hardware in TORCH takes care of maintaining the sequential state by buffering the side effects and results of boosted instructions until the conditional branch they are dependent upon is executed. Through the use of a trace-driven simulation, we found that a straightforward implementation of TORCH (including both simple hardware and a limited software instruction scheduling) competes with an aggressive, dynamic-scheduled superscalar processor on non-numerical applications.

With this encouraging data, we proceeded to develop a more complete software scheduling algorithm to exploit the instruction-level parallelism in the non-numerical code. We plan to implement this algorithm in a real compiler that we are building here at Stanford. The implementation is designed to be general enough to provide for experimentation with a variety of hardware configurations. It will be able to generate code for VLIW-like processors and for TORCH processors. The algorithm is similar to trace scheduling in that it takes a more global look at the program during code generation and scheduling in order to better utilize the instruction-level parallelism in a program. The algorithm differs from the original definition of trace scheduling in that it uses more control flow and control dependence information in order to limit the amount of code expansion. We plan to evaluate the scheduling algorithm and different hardware configurations through trace-driven simulation. As an aside, this evaluation system will also be able to be configured to provide data on the effectiveness of software scheduling in dynamically-scheduled superscalar processors.

6. Computer-Aided Design

6.1 Bisim

Work on Bisim is an attempt to construct a switch level simulator for digital circuits whose capabilities bridge the space between traditional switch level simulators and circuit simulators. Our principle goal is to create a switch-level simulator for MOS, ECL and BiCMOS circuits. However, we believe this framework will also allow the user to make a speed/accuracy tradeoff as well, allowing him to use different accuracies for different parts of the circuit.

Circuit simulation and switch level simulation have traditionally used disparate circuit analysis techniques to achieve different goals. Circuit simulators [1] are constructed to allow the accurate simulation of arbitrary circuits. They typically utilize nonlinear transistor models and place few restrictions on the circuit topology. Numerical integration is used to analyze the response of the circuit. Because the complexity of these algorithms is O(n^3) [2], circuit simulators can only simulate relatively small portions of an integrated circuit.

In contrast, switch level simulators take advantage of the fact that the full generality of a circuit simulator is unnecessary for predicting the first order behavior of most digital MOS circuits. In order to achieve increased simulation speed, switch level simulators [3] decompose the circuit into small clusters which are analyzed individually, restrict the topology of these clusters to transistor-capacitor trees, and restrict the transistor model to that of a switched resistor. The Elmore delay [4] [5] [6] is then used to estimate waveforms and delays. Analysis of this limited class of circuits can be very fast; up to three orders of magnitude faster than circuit simulation. Because the computational complexity of switch level algorithms is O(n) [6], these simulators can be applied to entire integrated circuits. However, there are frequently small portions of the circuit which must be simulated at the circuit level because the simplified switch models do not allow the accurate prediction of their behavior.

Recently, researchers [7] have extended the Elmore Delay and moment techniques to; 1) allow the computation of arbitrarily accurate waveform estimates through the use of higher order moments, and 2) allow the analysis of arbitrary linear networks including floating capacitors, inductors, and dependent sources in arbitrary topologies. Thus, Asymptotic Waveform Estimation extends the techniques originally developed for switch level simulation to the estimation of arbitrarily accurate responses of general linear circuits. These techniques have (apparently) been combined with piecewise linear transistor models to form a simulator, AWEsim [8], which possesses many of the capabilities of a traditional circuit simulator.

Unfortunately, the tree/link analysis used by AWEsim is not as efficient as the RC tree analysis techniques for the particular case of RC trees. The complexity of tree/link analysis applied to RC trees is O(n^2) [9] as opposed to O(n). This has lead to a recent attempt [9] to accelerate the analysis through the use of Path Tracing. However, although this work succeeds in reducing the number of multiplies to O(n), the number of additions remains O(n^2). Furthermore, RC tree analysis has been generalized [10] to handle multiple sources while retaining linear complexity. As described, the complexity of the Path Tracing algorithm is O(n^3) if there is a resistor to ground at each node.

Our approach is to implement the theoretical ideas in Asymptotic Waveform Estimation using an extension of the efficient tree analysis techniques developed for switch level simulation. We will then embed this delay analysis technique into a switch level framework to produce a simulator which can run almost as fast as switch level simulators for the simple device models and topologies required for most of a digital MOS circuit, but which can simultaneously simulate the more complex portions of the circuit at the cost of increased run time.

We will first assume that a digital circuit can be partitioned into channel connected "clusters" with identifiable inputs and outputs. It is assumed that the delay of a cluster can be analyzed independently of all other clusters once the input waveforms are known. The interaction between clusters is therefore restricted to take place via the inputs and outputs. An event driven scheduling mechanism will be used to simulate these interactions.

We initially restrict the topology of a cluster to be a transistor capacitor tree (possibly with multiple sources) because efficient algorithms exist for the computation of its moments. We will then show that the RC tree algorithms can be generalized to include trees of general linear three terminal networks (where the third terminal of each network is connected to ground). This allows us to utilize arbitrarily complex piecewise linear circuits to model transistors. The analysis of any particular circuit state still has complexity O(n), although we will have to perform the analysis multiple times as regions of linearity are crossed.

We will demonstrate the feasibility of the approach by creating the simulator and then applying it to the simulation of various MOS, ECL, and BiCMOS digital circuits.

References:

1. Nagel, L., *SPICE2: A Computer Program to Simulate Semiconductor Circuits*, University of California, Berkeley, Technical Report, ERL-520, May, 1975.

2. Chua, L. and Lin, P. Computer Aided Analysis of Electronic Circuits: Algorithms and Computational Techniques. 1975. Prentice-Hall.
3. Terman, C. J. *Simulation Tools for Digital LSI Design*. PhD, Massachusetts Institute of Technology, September, 1983.
4. Elmore, W. "The Transient Response of Damped Linear Networks with Particular Regard to Wideband Amplifiers," *Journal of Applied Physics*. 10, 55-63, January, 1948.
5. Penfield, P., Rubinstein, J. and Horowitz, M. "Signal Delay in RC Tree Networks," *IEEE Transactions on Computer-Aided Design of IC's*. CAD-2, (No. 3): 202-211, July, 1983.
6. Horowitz, M. A. *Timing Models for MOS Circuits*. PhD, Stanford University, December, 1983.
7. Pillage, L. and Rohrer, R. "Aymptotic Waveform Evaluation for Timing Analysis," *IEEE Transactions on Computer-Aided Design*. April, 1990.
8. Huang, X., Raghavan, V. and Rohrer, R. A., AWEsim: A Program for the Efficient Analysis of Linear(ized) Circuits, *International Conference on Computer-Aidedc Design*, November, 1990.
9. A Path Tracing Algorithm for Aymptotic Waveform Evaluation of Lumped RLC Circuit Delay Models. August, 1990.
10. Chu, C. Y. *Improved Models for Switch-Level Simulation*. PhD, Stanford University, November, 1988.
11. Rohrer, R. A. "Circuit Partitioning Simplified," *IEEE Transactions on Circuits and Systems*. 35, (No. 1): 2-5, January, 1988.

6.2 IRSIM

Our research consists of creating a set of CAD tools for the verification of MOS digital designs using incremental techniques. An incremental tool is characterized by a run time proportional to the size of a design change rather than the size of the entire design.

We have concentrated on two programs: incremental simulator, and incremental netlist extractor.

Our incremental simulator, IRSIM [1], attempts to decrease the time required to validate a design following a set of modifications by the user. The simulator takes advantage of the fact that, after an initial design, modifications have a low impact on the overall correctness of a design. This means that much of the computation performed in simulating a modified circuit are identical to those performed during previous simulations. By saving and reusing previous results, substantial savings in computation time can be achieved.

Until now, users were required to manually provide netlist-change commands to the simulator; a time consuming and error prone operation. To remedy this situation, we are adapting our layout editor, Magic [2], to perform incremental circuit extraction and automatically generate the netlist-change commands required by the simulator. Magic is a hierarchical layout editor that allows nearly arbitrary overlap between cells. Its extractor is hierarchical and supports a reduced notion of incrementalness by extracting each cell independently of its context so that only modified cells and its ancestors need to be re-extracted. Prior to simulation, the extracted (hierarchical) netlists need to be converted into a flat netlist.

The successful use of incremental techniques on large designs hinges on the problem of quickly identifying the modified portions of a design and communicating these changes to the incremental simulator. The problem is not as obvious as it seems, since in a hierarchical design system, even simple changes to the layout, such as deleting a piece of paint, may result in many changes to the underlying (flat) netlist. Conversely, large changes to the geometry, such as moving a subcircuit from one place to another or renaming a terminal, may result in no electrical changes to the netlist. The basic problem is that each tool operates on a different description of the same circuit; hierarchical mask geometry versus flat electrical netlists.

The required netlist changes could be identified by using Magic's extractor to extract the modified cells and then flatten the design and compare the resulting netlist to the original. Although this approach is possible, the overhead associated with flattening and comparing large netlists can quickly become prohibitive as the size of the design increases.

To solve this problem, our extractor uses a finer granularity than just cells: regions of change. This is accomplished by recording all areas modified by the user during an editing session and then re-extracting only these areas in their proper (flat) context for both the updated and the original network. The two netlists are then electrically compared against each other and the differences are reported to the simulator.

We have developed an extractor that is capable of extracting non-rectangular areas and (possibly) incomplete circuits while still maintaining the correct hierarchical position of each node or device in the flat netlist. Incomplete descriptions usually occur due to the overlap of one or more unmodified cells within a changed area, or when the change only consists of a device re-size.

The extracted netlist is represented as a graph where nodes and devices are represented as graph vertices, and the graph edges correspond to connections between nodes and devices. Representing netlists as graphs reduces the problem of comparing them to the well-known problem of graph isomorphism between 2 graphs [3-5]. Although the general problem is known to be NP-complete, for circuit (near planar) graphs several algorithms have been proposed to check for isomorphism. However, most of these algorithms either simply indicate whether the 2 graphs are isomorphic or just indicate the elements that aren't equivalent [6-9]. Our problem calls for finding the isomorphism function (the transformations needed to convert one graph into the other).

The netlist comparator is based on a graph isomorphism approach that uses graph-invariants but in addition uses the information available to circuit graphs such as transistor type, terminal connection (i.e. source or gate), and connections to corresponding nodes. These nodes are found by observing that any connection to a node outside a changed area must be the same in both netlists. These equivalent nodes provide the algorithm with a good starting point. The algorithm's goal is then to keep partitioning the 2 graphs until each vertex is uniquely identifiable and then determining whether that vertex has its counterpart in the other graph. Vertices (devices/nodes) found only in the current graph will be added, those appearing only in the old graph will be deleted.

In addition, in order to keep differences from propagating throughout the graph, differences are "fixed" as soon as they are identified. For example, a transistor whose gate and drain are connected to the same nodes in both graphs is not deleted but rather, the differing terminal is connected to the new corresponding node. This keeps the algorithm from diverging in the presence of minor differences; a common problem with netlist comparators.

Status:

- incremental extractor has been fully implemented and tested.

- netlist comparator is under development.

References:
1. Salz, A. and Horowitz, M. A. "IRSIM: An Incremental MOS Switch-Level Simulator," *DAC/ACM*. 173-178, June, 1989.
2. Scott, W. S. and Ousterhout, J. K. "Magic's Circuit Extractor," *DAC/ACM*. 286-292, June, 1985.
3. Read, R. and Corneil, D. "The Graph Isomorphism Disease," *Journal of Graph Theory*. 1, 339-363, 1977.
4. Corneil, D. "A Theoretical Analysis of Various Heuristics for the Graph Isomorphism Problem," *SIAM Journal of Computing*. 281-297, 1980.
5. Luks, E. "Isomorphism of Graphs of Bounded Valence can be Tested in Polynomial Time," *21st Foundations of Computer Science*. 42-49, 1980.
6. Spickelmier, R. and Newton, R. "WOMBAT: A New Netlist Comparison Program," 170-171, September, 1983.
7. Ebeling, C. and Zajieck, O. "Validating VLSI Circuit Layout by Wirelist Comparison," 172-173, September, 1983.
8. Reeves, D. and Irwin, M. "Fast Methods for Switch-Level Verification of MOS Circuits," *IEEE Transactions on CAD*. 3, 766-779, 1987.
9. Shiran, Y. "YNCC: A New Algorithm for Device-Level Comparison Between Two Isomorphic VLSI Circuits," 298-301, 1986.

6.3 Verification Techniques

Under other funding, we are developing programs to help analyze protocols and controllers that are used in hardware. Protocols and controllers are susceptible to subtle design errors which are difficult to detect and diagnose by simulation or prototyping. Our approach is to use state graphs both to model the actual behavior of implementations and to specify their desired behavior. We are applying these techniques to the DASH cache coherence protocol, in the hope that we can prove it correct or accelerate the debugging of the system.

We are constructing software tools that can compare finite-state representations of systems with specifications in the same form. The tool can derive the state graph from a program or other operational description. The analysis can consist of checking for a known interesting property (such as deadlock), making sure that a property (an invariant) is true in all reachable states, or comparing for consistency with a second finite state graph.

We are using two different approaches for exploring the state space. In the first approach, the protocol is described using condition/action rules (similar to guarded commands). The system state space is generated by simulating all firings for the rules, while saving all states in a large hash table. This method has the advantage that it is easy to implement, and the protocols are easy to describe. However, it suffers from the "state explosion problem" --- the protocol can generate too many states to fit in memory.

Our second approach is to use a symbolic representation of the state space, such as a boolean formula. The advantage of symbolic representations is that large sets often have simple structure which can be captured in a small symbolic form. We are currently using ordered binary decision diagrams, a widely-used representation of boolean functions for digital computer-aided design systems.

For a large system like DASH, the first step in verification is to reduce the description of the system so that it is still likely to exhibit any errors, but has relatively few states. For example, our current description has three clusters, three memory lines, and one bit of data in each memory line. We now have a description which models all of the basic memory transactions. Using the first method (hash table), this description generates more than 250,000 states. We have not yet discovered any bugs in the protocol, but we have found several unexpected events that were not covered in the documentation.

Our future plans are to add more details to the protocol description for use in the symbolic analyzer, and to check more complicated properties. For example, we will compare our description of the DASH behavior with a state machine describing the correct user-visible behavior (release consistency). We are also enhancing the tools, especially their ability to handle larger state spaces. We will be modifying the DASH description accordingly to take advantages of these improvements.

6.4 Partitioning of Functional Models

We have studied a technique aiming at multiple-chip synthesis from a a single high-level model in a Hardware Description Language (HDL). The partitioning of hardware functions in a chip set is crucial in achieving an efficient implementation. While hardware partitioning is dictated by the chip area limitations, it affects the performance of the overall system. The purpose of this research is to investigate computer-aided partitioning techniques that allow efficient implementation of hardware in multiple chips.

Unlike previous approaches, we use a partitioning technique performed at the functional abstraction level, where the digital hardware being designed is represented by a sequencing abstraction model capturing the operations to be performed and their dependencies. Such a functional model is a common abstraction in high-level synthesis, because it can be obtained by compiling a hardware description in a HDL and it forms a convenient data-structure for synthesis algorithms.

Our partitioning approach is motivated by the following reasons. We assume that the hardware being designed is synthesized from a high-level model in a HDL under a maximum timing constraint on the overall hardware latency. By using high-level synthesis techniques, the designer may try first to find a design configuration (i.e. binding and schedule) that satisfies the chip area and latency constraints. When such a structure cannot be found, then partitioning is used to overcome the area limitations while meeting the timing requirements. It is important to note that partitioning may introduce timing penalties, due to the inter-chip communication delays. For this reason, the designer will choose a design configuration that satisfies the latency constraint as a starting point for partitioning. Thus the search for a binding (that defines the hardware sharing) is done prior to partitioning, and it benefits the partitioning method in providing a starting point with an estimated area smaller than an unbound configuration.

This approach is important for hardware prototyping using programmable gate arrays that have a limited capacity in terms of gate count. By using the same functional model in a HDL, both a multi-chip prototype and the final implementation can be synthesized automatically. Bounds on the latency of the prototype are important to insure that accurate performance measures can be derived from it.

A major advantage of applying partitioning techniques at the functional abstraction level is that scheduling techniques can be applied concurrently to partitioning. Therefore, the overall latency of a partitioned structure can be readily evaluated, including the inter-chip communication delays. In this way, area-performance trade-offs can be exploited. Secondly, the functional model allows us to capture large hardware systems with fewer objects than at the logic netlist abstraction level. As a result, the partitioning algorithms are more efficient for large scale designs.

We have formulated the high-level partitioning problem as a constrained hypergraph partitioning problem and researched the application to this model of the Kernighan-Lin and the Simulated Annealing algorithms. A computer implementation in the program, *Vulcan,* has shown that the technique is viable.

6.5 Simulation

Initially, a simulator integration framework and algorithms were investigated. It was followed by a first prototype to obtain a running multi-level mixed-mode simulator on conventional workstations. The next step is to extend the prototype to run on parallel machines.

The framework of interest was a distributed discrete-event simulation within a set of communicating elements that exchange information through messages and distributed time. We started with the well-known time-stamped algorithms (Chandy and Misra) to reduce deadlock occurrence, increase concurrency, and reduce communication traffic overhead. The concepts of "intervals", rather than time stamps, are used to represent the period of time during which an event is valid in a simulation. With their use, there are situations in which a simulated element will not block while it would have otherwise if time-stamped messages were used. The use of labels on interval messages was introduced to allow messages to detect loops and obtain event scheduling optimizations. An interesting feature of intervals is that it allows simulation of future intervals in present time. These results were presented in the SCS Multiconference on Modeling and Simulation on Microcomputers [27].

A prototype to test and explore these concepts in a parallel multi-level mixed-mode simulation is under development. The simulators being integrated consist of THOR, a behavioral simulator for use with digital circuits at either the functional, register transfer or gate level, IRSIM, a switch-level simulator for MOS circuits, and SPICE, a general-purpose circuit-level simulator. This prototype is like a printed circuit board backplane where existing simulator programs are plugged in to run in parallel either at the same design level or at different design levels to function as a harmonic simulator. Basically, it consists of a kernel to handle the interface to simulators and coordinate their computation. It uses the concepts mentioned above for interval messages, and labeled messages within loops. Also, it handles conversion between design levels as in the case of logic signals and circuit signals. In this case, it uses known solutions as a threshold function between circuit to logic and a step/ramp function between logic to circuit.

A first prototype version was implemented and it is currently being tested. The goal is to test and explore the concepts mentioned above and to obtain a running multi-level mixed-mode simulation on conventional workstations. It consists of a modified version of the simulators being integrated and a common kernel that handles their interfaces and synchronization. This first prototype system is used for the verification of an adaptive signal processing system at both the chip and the board levels, and for the design of a complex CMOS Viterbi detection chip.

The second prototype version to be initiated will port this system to a multiprocessor system. This task will require a change of a localized and relatively small portion of the kernel that handles the communication between different instances of the simulators.

7. VLSI Design

7.1 Zero-Overhead Self-Timed Circuits

We have been looking at methods of building highest performance self-timed circuits. To accomplish this goal required finding a method to eliminate the control overhead that is normally associated with a self-timed design. This overhead arises from the fact that one usually needs to detect completion and then use this information to control the circuit. For most self-timed designs this control path ends up in the critical path.

We have developed a method of designing circuit with zero-overhead, precalculating the control information for each block. This method allows the control signals to enable a block before valid data arise at its inputs. The cost of this technique is a slight increase in hardware, since there needs to be enough stages between where the control signal is generated and used, so that the control delay can be hidden. To test this technique we designed and fabricated a 54bit self-timed divider. The chip used five stages connected in a self-timed ring, and was fabricated in a 1.2u CMOS technology by MOSIS. The 7 mm2 chip calculates a new quotient bit every 2.7ns at room temperature and 5V. Since the ring is self-timed, it is easy to measure its performance -- on connects, the power and simply measures the loop time.

7.2 BiCMOS SRAMS

The requirement of ECL-CMOS level conversion slows the access of traditional BiCMOS static RAMS. The CMOS-storage, emitter-access (CSEA) memory cell overcomes this limitation by placing a bipolar transistor into the memory cell itself. This cell allows a read path consisting entirely of low (ECL-ish) voltage swings which may be implemented using fairly standard ECL circuit techniques borrowed from bipolar static RAMS. As an outgrowth of our work on a sub-4 ns, 64Kbit BiCMOS static RAM [33], we have identified two areas which merit additional attention. We have been looking into techniques to reduce access-time penalties due to supply noise coupling into the bit line sense circuits and the feasibility of the CSEA cell in embedded (wide access path) cache memories for BiCMOS microprocessors.

Traditional static RAM design avoid problems with power supply noise slowing sense times by using fully-differential circuit techniques which can make the noise look common-mode to the sensing circuitry. However, the CSEA cell only provides one bit line for reading, so single-ended circuit techniques are used for sensing. These problems are exacerbated by heavily data-dependent supply coupling into the bit lines; the base-emitter capacitance of each emitter-follower forming the wired-or bit line is tightly coupled to either the positive supply (Vcc) or the negative supply (Vee), depending on the data stored in the cell. Note that this means the amount of charge dumped onto the bit line is dependent on the data stored in the *unselected* cells on that bit line. This prevents the use of a fully-differential CSEA cell (two followers and two bit lines) to make the noise coupling common-mode. It also limits the use of replica techniques in the reference-generating circuitry (as used in dynamic RAM design) because the replica circuit needs to have the same values stored in its unselected cells in order to experience the same coupling.

Our design senses the selected cell's value by comparing it to a reference voltage by turning the bit line into the shared node of a large ECL OR gate; the selected cell's follower and a sense device with the reference tied to its base form a differential pair. The presence or absence of current in this sense device is the signal read by the sense amp. In order to save power and circuitry, this sense amp is shared between many bit lines; the current from a sense device is summed with that from other sense devices (which should be zero if the associated bit line is unselected) at the emitter of a cascode-connected bipolar device (in order to reduce the voltage swing on a highly-capacitive node). A two-level network of these reduces the maximum nodal capacitance and hence speeds access.

Our approach to the supply noise issues relies on the large bit line current densities that the bipolar transistor in the memory cell allows. Simulations indicate that, worst case, we should be able to withstand substantial supply bounce without losing more than a quarter of our sense current. Design of the sense device's reference generator must take into account that supply noise coupling onto this reference is dependent on the values stored in *all* cells, since even unselected bit lines couple to this node through their sense devices' base-emitter capacitance. Preventing this reference from ever bouncing more than a selected bit line will avoid problems with excess/lost current due to reference bounce. Supply coupling issues affect the design of the cascode reference as well; because cascode trees without a selected bit line must not be allowed to generate substantial currents when their emitter node bounces, the reference generator is designed to track the emitter bounce of unselected trees. Finally, replica techniques are used to generate the reference voltage for comparison with the sense amp output; a dummy bit line with "average" coupling and half the normal selected current is fed into an equivalent cascode network.

Work on using the CSEA cell for embedded memories continues. A significant question is whether there is an alternative to such high bit line currents (for noise immunity), since this current will add up to very substantial power for very wide (hundreds of bits) access paths.

8. Publications, Presentations, Reports

1. Berlin, A. and Weise, D. "Compiling Scientific Code using Partial Evaluation," *IEEE Computer*. 23, (9): December, 1990.

2. Chow, F. and Hennessy, J. "The Priority-based Coloring Approach to Register Allocation," *IEEE Transactions on Programming Languages and Systems*. October, 1990. To be published.

3. Davis, H. and Goldschmidt, S., *Tango: A Multiprocessor Simulation and Tracing System*, Stanford University, Technical Report, CSL-90-436, July, 1990.

4. Dill, D. and Loewenstein, P., Verification of Multiprocessor Cache Protocol using Refinement Relations and Higher-Order Logic, Rutgers University, *Workshop on Computer-Aided Verification*, 1990.

5. Dill, D. and Loewenstein, P., Formal Verification of Cache Systems using Refinement Relations, IEEE, *International Conference on Computer Design*, 228-233, 1990.

6. Gharachorloo, K., Lenoski, D., Laudon, J., Gupta, A. and Hennessy, J., Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors, IEEE/ACM, *17th Annual International Conference on Computer Architecture*, Seattle, WA. May, 1990.

7. Gharachorloo, K., Gupta, A. and Hennessy, J. L., Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors, Submitted.

8. Gibbons, P. B. "A Synthesis of Parallel Algorithms." Asynchronous PRAM Algorithms. Reif ed. 1990 Morgan-Kaufmann. San Mateo.

9. Gibbons, P. B., Cache Support for the Asynchronous PRAM, ACM, *19th International Conference on Parallel Processing*, St. Charles, IL. August, 1990.

10. Gupta, A., Tucker, A. and Urushibara, S. "The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications," 1990. Submitted.

11. Gupta, A., Weber, W.-D. and Mowry, T., Reducing Memory and Traffic Requirements for Scalable Director-Based Cache Coherence Schemes, *International Conference on Parallel Processing*, 312-321, Pennsylvania State University. August, 1990.

12. Hayes, B., Anonymous One-Time Signatures and Flexible Untracable Electronic Cash, *Proceedings of AUSCRYPT*, 228-238, Sydney, Australia. August, 1990.

13. Katz, M. and Weise, D., Continuing into the Future: On the Interaction of Futures and First-Class Continuations. ACM, *Conference on Lisp and Functional Programming*, 176-184, Nice, France. June, 1990.

14. Lam, M. "Instruction Scheduling for Superscalar Architectures," *Annual Review*. 4, 173-201, 1990.

15. Lam, M. "The Software Pipelining Algorithm and Experimental Results," *Transactions on Programming Languages and Systems*. 1990. Submitted.

16. Lam, M. and Rinard, M., Course-Grain Parallel Programming in Jade, 1991. Submitted,

17. Lam, M., Rothberg, E. and Wolf, M., The Cache Performance and Optimizations of Blocked Algorithms. Submitted.

18. Lenoski, D., Laudon, J., Gharachorloo, K., Gupta, A. and Hennessy, J., The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor, IEEE, *17th Annual International Symposium on Computer Architecture*, 148-159, Seattle, WA. May, 1990.

19. Mowry, T. and Gupta, A. "Tolerating Latency Through Software-Controlled Prefetching in Scalable Shared-Memory Multiprocessors," 1990. Submitted.

20. Rothberg, E. and Gupta, A. "A Comparative Evaluation of Nodal and Supernodal Parallel Sparse Matrix Factorization: Detailed Simulation Results," 1990. Submitted.

21. Rothberg, E. and Gupta, A. "Efficient Sparce Matrix Factorization on High-Performance Workstations--Exploiting the Memory Hierarchy," *ACM Transactions on Mathematical Software*. 1990. To appear.

22. Rothberg, E. and Gupta, A., *Parallel ICCCG on a Hierarchical Memory Multiprocessor--Addressing the Triangular Solve Bottleneck*, Stanford University, Computer Systems Laboratory, Technical Report, CSL-90-449, September, 1990.

23. Rothberg, E. and Gupta, A., Techniques for Improving the Performance of Sparse Matrix Factorization on Multiprocessor Workstations, IEEE Computer Society, *Supercomputing '90*, New York, NY. November, 1990.

24. Ruf, E. and Weise, D., LogScheme: Integrating Logic Programming into Scheme, *Lisp and Symbolic Computation*, 1990.

25. Singh, J. and Hennessy, J. L. "Parallelizing an Ocean Simulation Program: Experience, Results and Implications," 1990. Submitted.

26. Smith, M., Lam, M. and Horowitz, M., Boosting Beyond Static Scheduling in a Superscalar Processor, IEEE/ACM, *17th Annual International Conference on Computer Architecture*, 344-354, Seattle, WA. May, 1990.

27. Todesco, A. and Meng, T., Interval Methods for Distributed Simulation systems, *Proceedings on SCS Multiconference on Modeling and Simulation on Microcomputers*, San Diego, CA. January, 1990.

28. Torrellas, J., Hennessy, J. and Weil, T., Analysis of Critical Architectural and Program Parameters in a Hierarchical Shared-Memory Multiprocessor, ACM, *Sigmetrics*, Denver, CO. May, 1990.

29. Torrellas, J., Lam, M. and Hennessy, J. L., Shared Data Placement Optimizations to Reduce Multiprocessor Cache Miss Rates, *International Conference on Parallel Processing*, 266-270, Pennsylvania State University. August, 1990.

30. Torrellas, J., Lam, M. and Hennessy, J. L. "Measurement, Analysis, and Improvement of the Cache Behavior of Shared Data in Cache Coherent Multiprocessors," 1990. Submitted.

31. Weise, D. "Formal Verification of MOS Circuits," *IEEE Transactions on Computer-Aided Design*. 9, (4): 341-351, 1990.

32. Weise, D., *Graphs as an Intermediate Representation for Partial Evaluation*, Stanford University, Computer Systems Laboratory, Technical Report, CSL-90-421, March, 1990.

33. Wingard, D., Stark, D., Horowitz, M. and Davis, H., *Circuit Techniques for Large CSEA SRAMs*, Stanford University, Technical Report, The Stanford BiCMOS Project, September, 1990. pp. 131-136.

34. Wolf, M. and Lam, M., An Algorithmic Approach to Compound Loop Transformations, *3rd Workshop on Languages and Compilers for Parallel Computing*, August, 1990.

9. Project Staff

Faculty:

John Hennessy Principal Investigator	jlh@vsop.stanford.edu	415/725-3712
Mark Horowitz Associate Investigator	horowitz@chroma.stanford.edu	415/725-3707
Anoop Gupta	ag@pepper.stanford.edu	415/725-3716
Monica Lam	lam@k2.stanford.edu	415/725-3714
Giovanni DeMicheli	nanni@mojave.stanford.edu	415/725-3632
Daniel Weise	daniel@mojave.stanford.edu	415/725-3711
Teresa Meng	meng@tilden.stanford.edu	415/725-3636
David Dill	dill@amadeus.stanford.edu	415/725-3642

Research Staff:
M. Ganapathi
Phil Gibbons
Charlie Orgish
David Nakahira
Laura Schrager

Graduate Students:
Saman Amarasinghe
Jennifer Anderson
Rohit Chandra
Tom Chanak
Helen Davis
Andrew Erlichson
Kourosh Gharachorloo
Aaron Goldberg
Steve Goldschmidt
Truman Joe
Lydia Kavraki
Jim Laudon
Dan Lenoski
John Maneatis
Margaret Martonosi
Dror Maydan
Scott McFarling
Arul Menezes
Todd Mowry
Jason Nieh
Karen Pieper
Steve Richardson
Martin Rinard
Ed Rothberg
Arturo Salz
Dan Scales
Rich Simoni
JP Singh
Mike Smith
Larry Soule
Don Stark
Luis Stevens
Steve Tjiang
Anthony Todesco
Josep Torrellas
Andrew Tucker
Wolf Weber
Ted Williams
Malcolm Wing
Drew Wingard
Michael Wolf

The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor

Daniel Lenoski, James Laudon, Kourosh Gharachorloo,
Anoop Gupta, and John Hennessy

Computer Systems Laboratory
Stanford University, CA 94305

Abstract

DASH is a scalable shared-memory multiprocessor currently being developed at Stanford's Computer Systems Laboratory. The architecture consists of powerful processing nodes, each with a portion of the shared-memory, connected to a scalable interconnection network. A key feature of DASH is its distributed directory-based cache coherence protocol. Unlike traditional snoopy coherence protocols, the DASH protocol does not rely on broadcast; instead it uses point-to-point messages sent between the processors and memories to keep caches consistent. Furthermore, the DASH system does not contain any single serialization or control point. While these features provide the basis for scalability, they also force a reevaluation of many fundamental issues involved in the design of a protocol. These include the issues of correctness, performance and protocol complexity. In this paper, we present the design of the DASH coherence protocol and discuss how it addresses the above issues. We also discuss our strategy for verifying the correctness of the protocol and briefly compare our protocol to the IEEE Scalable Coherent Interface protocol.

1 Introduction

The limitations of current uniprocessor speeds and the ability to replicate low cost, high-performance processors and VLSI components have provided the impetus for the design of multiprocessors which are capable of scaling to a large number of processors. Two major paradigms for these multiprocessor architectures have developed, *message-passing* and *shared-memory*. In a message-passing multiprocessor, each processor has a local memory, which is only accessible to that processor. Interprocessor communication occurs only through explicit message passing. In a shared-memory multiprocessor, all memory is accessible to each processor. The shared-memory paradigm has the advantage that the programmer is not burdened with the issues of data partitioning, and accessibility of data from all processors simplifies the task of dynamic load distribution. The primary advantage of the message passing systems is the ease with which they scale to support a large number of processors. For shared-memory machines providing such scalability has traditionally proved difficult to achieve.

We are currently building a prototype of a scalable shared-memory multiprocessor. The system provides high processor performance and scalability though the use of coherent caches and a directory-based coherence protocol. The high-level organization of the prototype, called DASH (Directory Architecture for SHared memory) [17], is shown in Figure 1. The architecture consists of a number of processing nodes connected through a high-bandwidth low-latency interconnection network. The physical memory in the machine is distributed among the nodes of the multiprocessor, with all memory accessible to each node. Each processing node, or *cluster*, consists of a small number of high-performance processors with their individual caches, a portion of the shared-memory, a common cache for pending remote accesses, and a directory controller interfacing the cluster to the network. A bus-based snoopy scheme is used to keep caches coherent within a cluster, while inter-node cache consistency is maintained using a distributed directory-based coherence protocol.

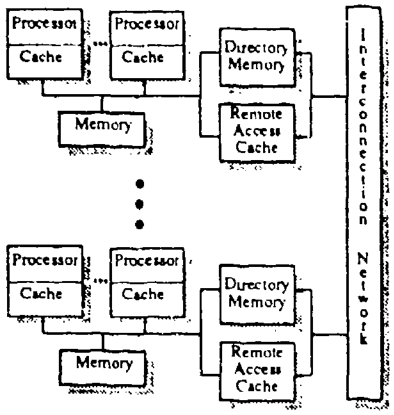

Figure 1: General architecture of DASH.

The concept of directory-based cache coherence was first proposed by Tang [20] and Censier and Feautrier [6]. Subsequently, it has been been investigated by others ([1],[2] and [23]). Building on this earlier work, we have developed a new directory-based cache-coherence protocol which works with distributed directories and the hierarchical cluster configuration of DASH. The protocol also integrates support for efficient synchronization operations using the directory. Furthermore, in designing the machine we have addressed many of the issues left unresolved by earlier work.

In DASH, each processing node has a directory memory corresponding to its portion of the shared physical memory. For each memory block, the directory memory stores the identities of all remote nodes caching that block. Using the directory memory, a node writing a location can send point-to-point invalidation or update messages to those processors that are actually caching that block. This is in contrast to the invalidating broadcast required by the snoopy protocol. The scalability of DASH depends on this ability to avoid broadcasts. Another important attribute of the directory-based protocol is that it does not depend on any specific interconnection network topology. As a result, we can readily use any of the low-latency scalable networks, such as meshes or hypercubes, that were originally developed for message-passing machines [7].

While the design of bus-based snoopy coherence protocols is reasonably well understood, this is not true of distributed directory-based protocols. Unlike snoopy protocols, directory-based schemes do not have a single serialization point for all memory transactions. While this feature is responsible for their scalability, it also makes them more complex and forces one to rethink how the protocol should address the fundamental issues of correctness, system performance, and complexity.

The next section outlines the important issues in designing a cache coherence protocol. Section 3 gives an overview of the DASH hardware architecture. Section 4 describes the design of the DASH coherence protocol, relating it to the issues raised in section 2. Section 5 outlines some of the additional operations supported beyond the base protocol, while Section 6 discusses scaling the directory structure. Section 7 briefly describes our approach to verifying the correctness of the protocol. Section 8 compares the DASH protocol with the proposed IEEE-SCI (Scalable Coherent Interface) protocol for distributed directory-based cache coherence. Finally, section 9 presents conclusions and summarizes the current status of the design effort.

2 Design Issues for Distributed Coherence Protocols

The issues that arise in the design of any cache coherence protocol and, in particular, a distributed directory-based protocol, can be divided into three categories: those that deal with correctness, those that deal with the performance, and those related to the distributed control of the protocol.

2.1 Correctness

The foremost issue that any multiprocessor cache coherence protocol must address is correctness. This translates into requirements in three areas:

Memory Consistency Model: For a uniprocessor, the model of a correct memory system is well defined. Load operations return the last value written to a given memory location. Likewise, store operations bind the value returned by subsequent loads of the location until the next store. For multiprocessors, however, the issue is more complex because the definitions of "last value written", "subsequent loads" and "next store" become less clear as there may be multiple processors reading and writing a location. To resolve this difficulty a number of memory consistency models have been proposed in the literature, most notably, the sequential and weak consistency models [8]. Weaker consistency models attempt to loosen the constraints on the coherence protocol while still providing a reasonable programming model for the user. Although most existing systems utilize a relatively strong consistency model, the larger latencies found in a distributed system favor the less constrained models.

Deadlock: A protocol must also be deadlock free. Given the arbitrary communication patterns and finite buffering within the memory system there are numerous opportunities for deadlock. For example, a deadlock can occur if a set of transactions holds network and buffer resources in a circular manner, and the consumption of one request requires the generation of another request. Similarly, lack of flow control in nodes can cause requests to back up into the network, blocking the flow of other messages that may be able to release the congestion.

Error Handling: Another issue related to correctness is support for data integrity and fault tolerance. Any large system will exhibit failures, and it is generally unacceptable if these failures result in corrupted data or incorrect results without a failure indication. This is especially true for parallel applications where algorithms are more complex and may contain some nondeterminism which limits repeatability. Unfortunately, support for data integrity and fault-tolerance within a complex protocol that attempts to minimize latency and is executed directly by hardware is difficult. The protocol must attempt to balance the level of data integrity with the increase in latency and hardware complexity. At a minimum, the protocol should be able to flag all detectable failures, and convey this information to the processors affected.

2.2 Performance

Given a protocol that is correct, performance becomes the next important design criterion. The two key metrics of memory system performance are latency and bandwidth.

Latency: Performance is primarily determined by the latency experienced by memory requests. In DASH, support for cachable shared data provides the major reduction in latency. The latency of write misses is reduced by using write buffers and by the support of the release consistency model. Hiding the latency for read misses is usually more critical since the processor is stalled until data is returned. To reduce the latency for read misses, the protocol must minimize the number of inter-cluster messages needed to service a miss and the delay associated with each such message.

Bandwidth: Providing high memory bandwidth that scales with the number of processors is key to any large system. Caches and distributed memory form the basis for a scalable, high-bandwidth memory system in DASH. Even with distributed memory, however, bandwidth is limited by the serialization of requests in the memory system and the amount of traffic generated by each memory request.

Servicing a memory request in a distributed system often requires several messages to be transmitted. For example, a message to access a remote location generates a reply message containing the data, and possibly other messages invalidating remote caches. The component with the largest serialization in this chain limits the maximum throughput of requests. Serialization affects performance by increasing the queuing delays, and thus the latency, of memory requests. Queueing delays can become critical for locations that exhibit a large degree of sharing. A protocol should attempt to minimize the service time at all queuing centers. In particular, in a distributed system no central resources within a node should be blocked while inter-node communication is taking place to service a request. In this way serialization is limited only by the time of local, intra-node operations.

The amount of traffic generated per request also limits the effective throughput of the memory system. Traffic seen by the global interconnect and memory subsystem increases the queueing for these shared resources. DASH reduces traffic by providing coherent caches and by distributing memory among the processors. Caches filter many of the requests for shared data while grouping memory with processors removes private references if the corresponding memory is allocated within the local cluster. At the protocol level, the number of messages required to service different types of memory requests should be minimized, unless the extra messages directly contribute to reduced latency or serialization.

2.3 Distributed Control and Complexity

A coherence protocol designed to address the above issues must be partitioned among the distributed components of the multiprocessor. These components include the processors and their caches, the directory and main memory controllers, and the interconnection network. The lack of a single serialization point, such as a bus, complicates the control since transactions do not complete atomically. Furthermore, multiple paths within the memory system and lack of a single arbitration point within the system allow some operations to complete out of order. The result is that there is a rich set of interactions that can take place between different memory and coherence transactions. Partitioning the control of the protocol requires a delicate balance between the performance of the system and the complexity of the components. Too much complexity may effect the ability to implement the protocol or ensure that the protocol is correct.

3 Overview of DASH

Figure 2 shows a high-level picture of the DASH prototype we are building at Stanford. In order to manage the size of the prototype design effort, a commercial bus-based multiprocessor was chosen as the processing node. Each node (or *cluster*) is a Silicon Graphics POWER Station 4D/240 [4]. The 4D/240 system consists of four high-performance processors, each connected to a 64 Kbyte first-level instruction cache, and a 64 Kbyte write-through data cache. The 64 Kbyte data cache interfaces to a 256 Kbyte second-level write-back cache through a read buffer and a 4 word deep write-buffer. The main purpose of this second-level cache is to convert the write-through policy of the first-level to a write-back policy, and to provide the extra cache tags for bus snooping. Both the first and second-level caches are direct-mapped.

In the 4D/240, the second-level caches are responsible for bus snooping and maintaining consistency among the caches in the cluster. Consistency is maintained using the Illinois coherence protocol [19], which is an invalidation-based ownership protocol. Before a processor can write to a cache line, it must first acquire exclusive ownership of that line by requesting that all other caches invalidate their copy of that line. Once a processor has exclusive ownership of a cache line, it may write to that line without consuming further bus cycles.

The memory bus (MPBUS) of the 4D/240 is a pipelined synchronous bus, supporting memory-to-cache and cache-to-cache transfers of 16 bytes every 4 bus clocks with a latency of 6 bus clocks. While the MPBUS is pipelined, it is not a split transaction bus. Consequently, it is not possible to efficiently interleave long duration remote transactions with the short duration local

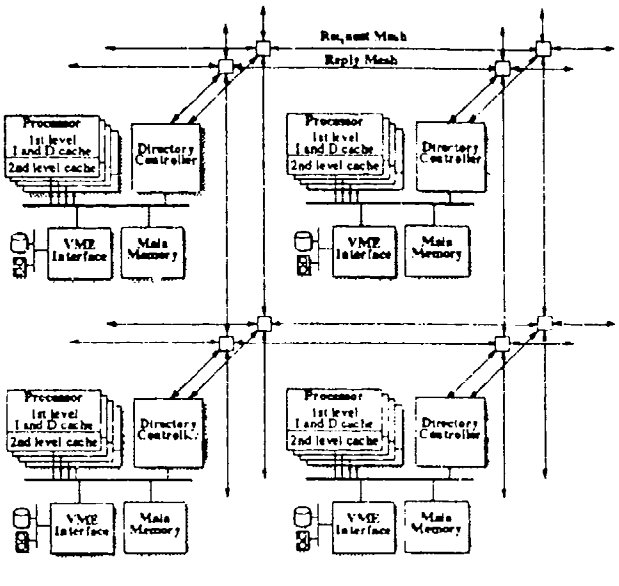

Figure 2: Block diagram of sample 2 x 2 DASH system.

transactions. Since this ability is critical to DASH, we have extended the MPBUS protocol to support a retry mechanism. Remote requests are signaled to retry while the inter-cluster messages are being processed. To avoid unnecessary retries the processor is masked from arbitration until the response from the remote request has been received. When the response arrives, the requesting processor is unmasked, retries the request on the bus, and is supplied the remote data.

A DASH system consists of a number of modified 4D/240 systems that have been supplemented with a directory controller board. This directory controller board is responsible for maintaining the cache coherence across the nodes and serving as the interface to the interconnection network.

The directory board is implemented on a single printed circuit board and consists of five major subsystems as shown in Figure 3. The *directory controller* (DC) contains the directory memory corresponding to the portion of main memory present within the cluster. It also initiates out-bound network requests and replies. The *pseudo-CPU* (PCPU) is responsible for buffering incoming requests and issuing such requests on the cluster bus. It mimics a CPU on this bus on behalf of remote processors except that responses from the bus are sent out by the directory controller. The *reply controller* (RC) tracks outstanding requests made by the local processors and receives and buffers the corresponding replies from remote clusters. It acts as memory when the local processors are allowed to retry their remote requests. The *network interface* and the local portion of the network itself reside on the directory card. The interconnection network consists of a pair of meshes. One mesh is dedicated to the request messages while the other handles replies. These meshes utilize *wormhole routing* [9] to minimize latency. Finally, the board contains *hardware monitoring logic* and miscellaneous control and status registers. The monitoring logic samples a variety of directory board and bus events from which usage and performance statistics can be derived.

The directory memory is organized as an array of directory

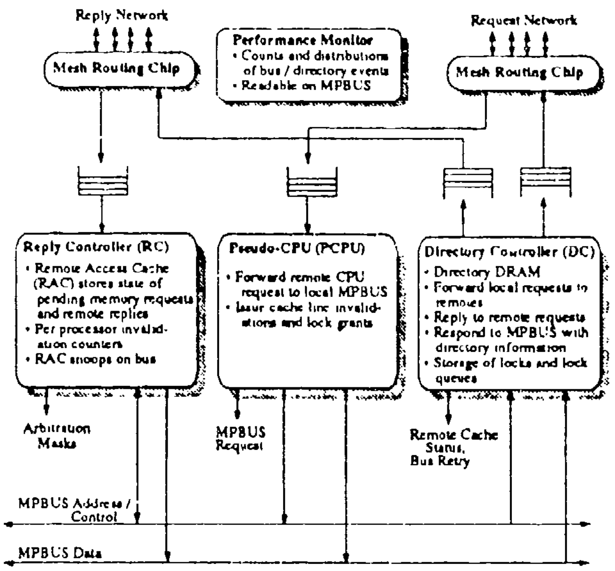
Reply Network
Mesh Routing Chip
Performance Monitor
• Counts and distributions of bus / directory events
• Readable on MPBUS
Request Network
Mesh Routing Chip
Reply Controller (RC)
• Remote Access Cache (RAC) stores state of pending memory requests and remote replies
• Per processor invalidation counters
• RAC snoops on bus
Pseudo-CPU (PCPU)
• Forward remote CPU request to local MPBUS
• Issue cache line invalidations and lock grants
Directory Controller (DC)
• Directory DRAM
• Forward local requests to remotes
• Reply to remote requests
• Respond to MPBUS with directory information
• Storage of locks and lock queues
Arbitration Masks
MPBUS Request
Remote Cache Status, Bus Retry
MPBUS Address / Control
MPBUS Data

Figure 3: Directory board block diagram.

entries. There is one entry for each memory block. The directory entries used in the prototype are identical to that originally proposed in [6]. They are composed of a single state bit together with a bit vector of pointers to clusters. The state bit indicates whether the clusters have a read (shared) or read/write (dirty) copy of the data. The bit vector contains a bit for each of the sixteen clusters supported in the prototype. Associating the directory with main memory allows the directory to be built with the same DRAM technology as main memory. The DC accesses the directory memory on each MPBUS transaction along with the access to main memory. The directory information is combined with the type of bus operation, address, and result of the snooping within the cluster to determine what network messages and bus controls the DC will generate.

The RC maintains its state in the *remote access cache* (RAC). The functions of the RAC include maintaining the state of currently outstanding requests, buffering replies from the network and supplementing the functionality of the processors' caches. The RAC is organized as a snoopy cache with augmented state information. The RAC's state machines allow accesses from both the network and the cluster bus. Replies from the network are buffered in the RAC and cause the waiting processor to be released for bus arbitration. When the released processor retries the access the RAC supplies the data via a cache-to-cache transfer.

3.1 Memory Consistency in DASH

As stated in Section 2, the correctness of the coherence protocol is a function of the memory consistency model adopted by the architecture. There is a whole spectrum of choices for the level of consistency to support directly in hardware. At one end is the *sequential consistency* model [16] which requires the execution of the parallel program to appear as some interleaving of the execution of the parallel processes on a sequential machine. As one moves towards weaker models of consistency, performance gains are made at the cost of a more complex programming model for the user.

The base model of consistency provided by the DASH hardware is called *release consistency*. Release consistency [10] is an extension of the weak consistency model first proposed by Dubois, Scheurich and Briggs [8]. The distinguishing characteristics of release consistency is that it allows memory operations issued by a given processor to be observed and complete out of order with respect to the other processors. The ordering of operations is only preserved before "releasing" synchronization operations or explicit ordering operations. Release consistency takes advantage of the fact that while in a critical region a programmer has already assured that no other processor is accessing the protected variables. Thus, updates to these variables can be observed by other processors in arbitrary order. Only before the lock release at the end of the region does the hardware need to guarantee that all operations have completed. While release consistency does complicate programming and the coherence protocol, it can hide much of the overhead of write operations.

Support for release consistency puts several requirements on the system. First, the hardware must support a primitive which guarantees the ordering of memory operations at specific points in a program. Such fence [5, 10] primitives can then be placed by software before releasing synchronization points in order to implement release consistency. DASH supports two explicit fence mechanisms. A *full-fence* operation stalls the processor until all of its pending operations have been completed, while a *write-fence* simply delays subsequent write-operations. A higher performance implementation of release consistency includes implicit fence operations within the releasing synchronization operations themselves. DASH supports such synchronization operations yielding release consistency as its base consistency model. The explicit fence operations in DASH then allow the user or compiler to synthesize stricter consistency models if needed.

The release consistency model also places constraints on the base coherence protocol. First, the system must respect the local dependencies generated by the memory operations of a single processor. Second, all coherence operations, especially operations related to writes, must be acknowledged so that the issuing processor can determine when a fence can proceed. Third, any cache line owned with pending invalidations against it can not be shared between processors. This prevents the new processor from improperly passing a fence. If sharing is allowed then the receiving processor must be informed when all of the pending invalidates have been acknowledged. Lastly, any operations that a processor issues after a fence operation may not become visible to any other processor until all operations preceding the fence have completed.

4 The DASH Cache Coherence Protocol

In our discussion of the coherence protocol, we use the following naming conventions for the various clusters and memories involved in any given transaction. A *local cluster* is a cluster that contains the processor originating a given request, while the *home cluster* is the cluster which contains the main memory and directory for a given physical memory address. A *remote cluster* is any other cluster. Likewise, *local memory* refers to the main memory associated with the local cluster while *remote memory* is any memory whose home is not the local.

The DASH coherence protocol is an invalidation-based ownership protocol. A memory block can be in one of three states as indicated by the associated directory entry: (i) *uncached-remote*, that is not cached by any remote cluster; (ii) *shared-remote*, that is cached in an unmodified state by one or more remote clusters; or (iii) *dirty-remote*, that is cached in a modified state by a single remote cluster. The directory does not maintain information concerning whether the home cluster itself is caching a memory block because all transactions that change the state of a memory block are issued on the bus of the home cluster, and the snoopy bus protocol keeps the home cluster coherent. While we could have chosen not to issue all transactions on the home cluster's bus this would had an insignificant performance improvement since most requests to the home also require an access to main memory to retrieve the actual data.

The protocol maintains the notion of an *owning cluster* for each memory block. The owning cluster is nominally the home cluster. However, in the case that a memory block is present in the dirty state in a remote cluster, that cluster is the owner. Only the owning cluster can complete a remote reference for a given block and update the directory state. While the directory entry is always maintained in the home cluster, a dirty cluster initiates all changes to the directory state of a block when it is the owner (such update messages also indicate that the dirty cluster is giving up ownership). The order that operations reach the owning cluster determines their global order.

As with memory blocks, a cache block in a processor's cache may also be in one of three states: invalid, shared, and dirty. The shared state implies that there may be other processors caching that location. The dirty state implies that this cache contains an exclusive copy of the memory block, and the block has been modified.

The following sections outline the three primitive operations supported by the base DASH coherence protocol: read, read-exclusive and write-back. We also discuss how the protocol responds to the issues that were brought up in Section 2 and some of the alternative design choices that were considered. We describe only the normal flow for the memory transactions in the following sections, exception cases are covered in section 4.6.

4.1 Read Requests

Memory read requests are initiated by processor load instructions. If the location is present in the processor's first-level cache, the cache simply supplies the data. If not present, then a cache fill operation must bring the required block into the first-level cache. A fill operation first attempts to find the cache line in the processor's second-level cache, and if unsuccessful, the processor issues a read request on the bus. This read request either completes locally or is signaled to retry while the directory board interacts with the other clusters to retrieve the required cache line. The detailed flow for a read request is given in Figure 7 in the appendix.

The protocol tries to minimize latency by using cache-to-cache transfers. The local bus can satisfy a remote read if the given line is held in another processor's cache or the remote access cache (RAC). The four processor caches together with the RAC form a five-way set associative (1.25 Mbyte) cluster cache. The effective size of this cache is smaller than a true set associative cache because the entries in the caches need not be distinct. The check for a local copy is initiated by the normal snooping when the read is issued on the bus. If the cache line is present in the shared state then the data is simply transferred over the bus to the requesting processor and no access to the remote home cluster is needed. If the cache line is held in a dirty state by a local processor, however, something must be done with the ownership of the cache line since the processor supplying the data goes to a shared state in the Illinois protocol used on the cluster bus. The two options considered were to: (i) have the directory do a sharing write-back to the home cluster; and (ii) have the RAC take ownership of the cache line. We chose the second option because it permits the processors within a cluster to read and write a shared location without causing traffic in the network or home cluster.

If a read request cannot be satisfied by the local cluster, the processor is forced to retry the bus operation, and a request message is sent to the home cluster. At the same time the processor is masked from arbitration so that it does not tie up the local bus. Whenever a remote request is sent by a cluster, a RAC entry is allocated to act as a placeholder for the reply to this request. The RAC entry also permits merging of requests made by the different processors within the same cluster. If another request to the same memory block is made, a new request will not be sent to the home cluster; this reduces both traffic and latency. On the other hand, an access to a different memory block, which happens to map to a RAC entry already in use, must be delayed until the pending operation is complete. Given that the number of active RAC entries is small the benefit of merging should outweigh the potential for contention.

When the read request reaches the home cluster, it is issued on that cluster's bus. This causes the directory to look up the status of that memory block. If the block is in an uncached-remote or shared-remote state the directory controller sends the data over the reply network to the requesting cluster. It also records the fact that the requesting cluster now has a copy of the memory block. If the block is in the dirty-remote state, however, the read request is forwarded to the owning, dirty cluster. The owning cluster sends out two messages in response to the read. A message containing the data is sent directly to the requesting cluster, and a sharing writeback request is sent to the home cluster. The sharing writeback request writes the cache block back to memory and also updates the directory. The flow of messages for this case is shown in Figure 4.

As shown in Figure 4, any request not satisfied in the home cluster is forwarded to the remote cluster that has a dirty copy of the data. This reduces latency by permitting the dirty cluster to respond directly to the requesting cluster. In addition, this forwarding strategy allows the directory controller to simultaneously process many requests (i.e. to be multithreaded) without the added complexity of maintaining the state of outstanding requests. Serialization is reduced to the time of a single intra-cluster bus transaction. The only resource held while inter-cluster messages are being sent is a single entry in the originating cluster's RAC.

The downside of the forwarding strategy is that it can result in additional latency when simultaneous accesses are made to the same block. For example, if two read requests from different clusters are received close together for a line that is dirty remote, both will be forwarded to the dirty cluster. However, only the first one will be satisfied since this request will force the dirty cluster to lose ownership by doing a sharing writeback and changing its local state to read only. The second request will not find the dirty data and will be returned with a *negative acknowledge* (NAK) to its originating cluster. This NAK will force the cluster to retry its access. An alternative to the forwarding approach used by our protocol would have been to buffer the read request at the home cluster, have the home send

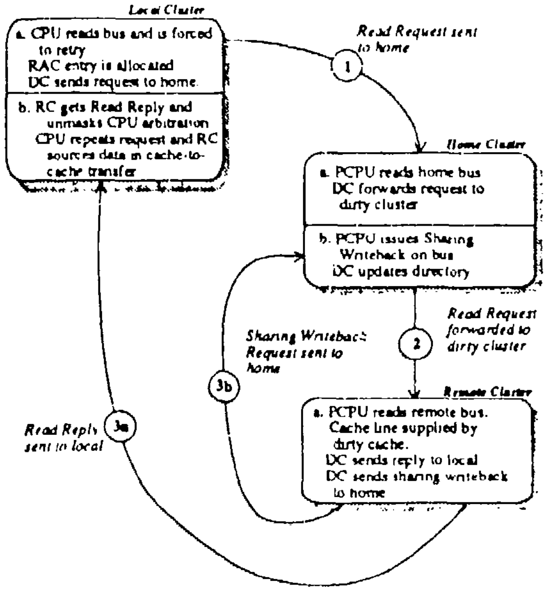
Local Cluster
a. CPU reads bus and is forced to retry
RAC entry is allocated
DC sends request to home.
b. RC gets Read Reply and unmasks CPU arbitration
CPU repeats request and RC sources data in cache-to-cache transfer
Read Request sent to home
1
Home Cluster
a. PCPU reads home bus
DC forwards request to dirty cluster
b. PCPU issues Sharing Writeback on bus
DC updates directory
Read Request forwarded to dirty cluster
2
Sharing Writeback Request sent to home
3b
Remote Cluster
a. PCPU reads remote bus.
Cache line supplied by dirty cache.
DC sends reply to local
DC sends sharing writeback to home
Read Reply sent to local
3a

Figure 4: Flow of Read Request to remote memory with directory in dirty-remote state.

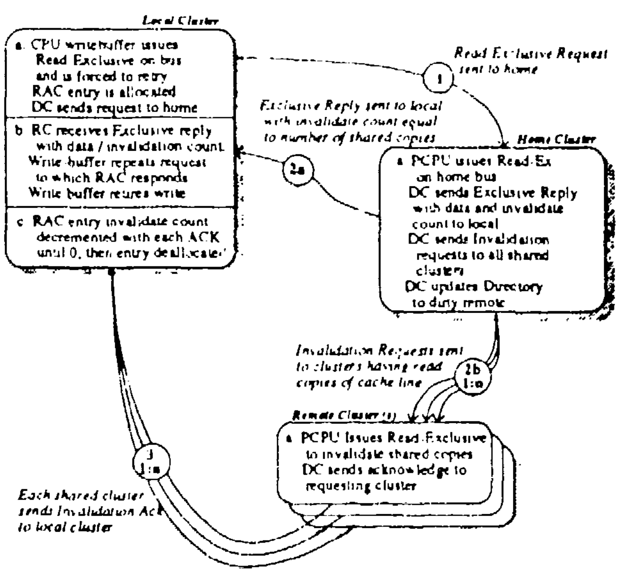
Local Cluster
a. CPU writebuffer issues Read Exclusive on bus and is forced to retry
RAC entry is allocated
DC sends request to home
b. RC receives Exclusive reply with data / invalidation count
Write-buffer repeats request to which RAC responds
Write buffer retires write
c. RAC entry invalidate count decremented with each ACK until 0, then entry deallocated
Read Exclusive Request sent to home
1
Exclusive Reply sent to local with invalidate count equal to number of shared copies
2a
Home Cluster
a. PCPU issues Read-Ex on home bus
DC sends Exclusive Reply with data and invalidate count to local
DC sends Invalidation requests to all shared clusters
DC updates Directory to dirty remote
Invalidation Requests sent to clusters having read copies of cache line
2b
1:n
Remote Cluster (i)
a. PCPU Issues Read-Exclusive to invalidate shared copies
DC sends acknowledge to requesting cluster
3
1:n
Each shared cluster sends Invalidation Ack to local cluster

Figure 5: Flow of Read-Exclusive Request to remote memory with directory in shared-remote state.

a flush request to the owning cluster, and then have the home send the data back to the originating cluster. We did not adopt this approach because it would have increased the latency for such reads by adding an extra network and bus transaction. Additionally, it would have required buffers in the directory to hold the pending transaction, or blocking subsequent accesses to the directory until the first request had been satisfied.

4.2 Read-Exclusive Requests

Write operations are initiated by processor store instructions. Data is written through the first-level cache and is buffered in a four word deep write-buffer. The second-level cache can retire the write if it has ownership of the line. Otherwise, a read-exclusive request is issued to the bus to acquire sole ownership of the line and retrieve the other words in the cache block. Obtaining ownership does not block the processor directly; only the write-buffer output is stalled. As in the case of read requests, cache coherence operations begin when the read-exclusive request is issued on the bus. The detailed flow of read-exclusive request is given in the appendix in Figure 9 and is summarized below.

The flow of a read-exclusive is similar to that of a read request. Once the request is issued on the bus, it checks other caches at the local cluster level. If one of those caches has that memory block in the dirty state (it is the owner), then that cache supplies the data and ownership and invalidates its own copy. If the memory block is not owned by the local cluster, a request for ownership is sent to the home cluster. As in the case of read requests, a RAC entry is allocated to receive the ownership and data.

At the home cluster, the read-exclusive request is echoed on the bus. If the memory block is in an uncached-remote or shared-remote state the data and ownership are immediately sent back over the reply network. In addition, if the block is in the shared-remote state, each cluster caching the block is sent an invalidation request. The requesting cluster receives the data as before, and is also informed of the number of invalidation acknowledge messages to expect. Remote clusters send invalidation acknowledge messages to the requesting cluster after completing their invalidation. As discussed in Section 3.1, the invalidation acknowledges are needed by the requesting processor to know when the store has been completed with respect to all processors. The RAC entry in the requesting cluster persists until all invalidation acknowledges have been received. The receipt of the acknowledges generally occurs after the processor itself has been granted exclusive ownership of the cache line and continued execution. Figure 5 depicts this shared-remote case.

If the directory indicates a dirty-remote state, then the request is forwarded to the owning cluster as in a read request. At the dirty cluster, the read-exclusive request is issued on the bus. This causes the owning processor to invalidate that block from its cache and to send a message to the requesting cluster granting ownership and supplying the data. In parallel, a request is sent to the home cluster to update ownership of the block. On receiving this message, the home sends an acknowledgment to the new owning cluster. This extra acknowledgment is needed because the requesting cluster (the new owning cluster) may give up ownership (e.g. due to a writeback) even before the home directory has received an ownership change message from the previous owner. If these messages reach the home out of order the directory will become permanently inconsistent. The extra acknowledgment guarantees that the new owner retain ownership until the directory has been updated.

Performance of the read and write operations is closely related to the speed of the MPBUS and the latency of inter-cluster communication. Figure 6 shows the latencies for various mem-

Read Operations	
Hit in 1st Level Cache	1 pclock
Fill from 2nd Level Cache	12 pclock
Fill from Local Cluster	22 pclock
Fill from Remote Cluster	61 pclock
Fill from Dirty Remote, Remote Home	80 pclock
Fill operations fetch 16 byte cache blocks and empty the write-buffer before fetching the read-miss cache block	
Write Operations	
Hit on 2nd Level Owned Block	3 pclock
Owned by Local Cluster	18 pclock
Owned in Remote Cluster	57 pclock
Owned in Dirty Remote, Remote Home	76 pclock
Write operations only stall the write-buffer, not the processor, while the fill is outstanding *Write delays assume Release Consistency (i.e. they do not wait for remote invalidates to be acknowledged)*	

Figure 6: Latency for various memory system operations in processor clocks. Each processor clock in the prototype is 40 ns.

ory operations in the DASH prototype assuming no network or bus contention. The figure illustrates the one-to-one relationship between the latency of an operation and its corresponding number of network hops and bus transactions. In DASH, the network and directory board overhead is roughly equal to the CPU overhead to initiate a bus transaction. Thus, if an intra-cluster bus transaction takes roughly 20 processor clocks then an inter-cluster transaction that involves two clusters, (i.e. three bus transactions) takes roughly 60 processor clocks, and a three cluster transaction takes 80 processor clocks.

4.3 Writeback Requests

A dirty cache line that is replaced must be written back to memory. If the home of the memory block is the local cluster, then the data is simply written back to main memory. If the home cluster is remote, then a message is sent to the remote home which updates the main memory and marks the block uncached-remote. The flow of a writeback operation is given in the appendix in Figure 8.

4.4 Bus Initiated Cache Transactions

CPU initiated transactions have been described in the preceding sections. The protocol also includes transitions made by the slave caches that are monitoring their respective buses. These transitions are equivalent to those in a normal snoopy bus protocol. In particular, a read operation on the bus will cause a dirty cache to supply data and change to a shared state. Dirty data will also be written back to main memory (or the RAC if remote). A read-exclusive operation on the bus will cause all other cached copies of the line to be invalidated. Note that when a valid line in the second-level cache is invalidated, the first-level cache is also invalidated so that the processor's second-level cache is a superset of the first-level cache.

4.5 Support for Memory Consistency

As discussed in section 3.1, DASH supports the release consistency model. Memory system latency is reduced because the semantics of release consistency allows the processor to continue after issuing a write operation. The write-buffer within the processor holds the pending operation, and the write-buffer is allowed to retire the write before the operation has completed with respect to all processors. The processor itself is allowed to continue while the write-buffer and directory controller are completing the previous operations. Ordering of memory accesses is only guaranteed between operations separated by a releasing synchronization operation or an explicit fence operation. Upon a write-fence (explicit or implicit), all previous read and write operations issued by this processor must have completed with respect to all processors before any additional write operations can become visible to other processors.

DASH implements a write fence by blocking a processor's access to its second-level cache and the MPBUS until all reads and writes it issued before the write fence have completed. This is done by stalling the write-fence (which is mapped to a store operation) in the processor's write-buffer. Guaranteeing that preceding reads and writes have been performed without imposing undue processor stalls is the challenge. A first requirement is that all invalidation operations must be acknowledged. As illustrated in Figure 5, a write operation to shared data can proceed after receiving the exclusive reply from the directory, but the RAC entry associated with this operation persists until all of the acknowledges are received by the reply controller (RC). Each RAC entry is tagged with the processor that is responsible for this entry and each processor has a dedicated counter in the RC which counts the total number of RAC entries in use by that processor. A write fence stalls until the counter for that processor is decremented to zero. At this point, the processor has no outstanding RAC entries, so all of its invalidation acknowledges must have been received.

We observe that simply using a per processor counter to keep track of the number of outstanding invalidations is not sufficient to support release consistency. A simple counter does not allow the processor cache to distinguish between dirty cache lines that have outstanding invalidates from those that do not. This results in another processor not being able to detect whether a line returned by a dirty cache has outstanding invalidates. The requesting processor could then improperly pass through a fence operation. Storing the pending invalidate count on a per cache line basis in the RAC, and having the RAC snoop bus transactions, allows cache lines with pending invalidates to be distinguished. The RAC forces a reject of remote requests to such blocks with a NAK reply. Local accesses are allowed, but the RAC adds the new processor to its entry for the line making this processor also responsible for the original invalidations. Write-back requests of a line with outstanding invalidations are blocked by having the RAC take dirty ownership of the cache block.

In the protocol, invalidation acknowledges are sent to the local cluster that initiated the memory request. An alternative would be for the home cluster to gather the acknowledges, and, when all have been received, send a message to the requesting cluster indicating that the request has been completed. We chose the former because it reduces the waiting time for completion of a subsequent fence operation by the requesting cluster and reduces the potential of a hot spot developing at the memory.

4.6 Exception Conditions

The description of the protocol listed above does not cover all of the conditions that the actual protocol must address. While enumerating all of the possible exceptions and protocol responses would require an overly detailed discussion, this section introduces most of the exception cases and gives an idea of how the protocol responds to each exception.

One exception case is that a request forwarded to a dirty cluster may arrive there to find that the dirty cluster no longer owns the data. This may occur if another access had previously been forwarded to the dirty cluster and changed the ownership of the block, or if the owning cluster performs a writeback. In these cases, the originating cluster is sent a NAK response and is required to reissue the request. By this time ownership should have stabilized and the request will be satisfied. Note that the reissue is accomplished by simply releasing the processor's arbitration mask and treating this as a new request instead of replying with data.

In very pathological cases, for example when ownership for a block is bouncing back and forth between two remote clusters, a requesting cluster (some third cluster) may receive multiple NAK's and may eventually time-out and return a bus error. While this is undesirable, its occurrence is very improbable in the prototype system and, consequently, we do not provide a solution. In larger systems this problem is likely to need a complete answer. One solution would be to implement an additional directory state which signifies that other clusters are queued for access. Only the first access for a dirty line would be forwarded while this request and subsequent requests are queued in the directory entry. Upon receipt of the next ownership change the directory can respond to all of the requests if they are for read only copies. If some are for exclusive access then ownership can be granted to each in turn on a pseudo-random basis. Thus, eventually all requests will be fulfilled.

Another set of exceptions arise from the multiple paths present in the system. In particular, the separate request and reply networks together with their associated input and output FIFO's and bus requesters imply that some messages sent between two clusters can be received out of order. The protocol can handle most of these misorderings because operations are acknowledged and out-of-order requests simple receive NAK responses. Other cases require more attention. For example, a read reply can be overtaken by an invalidate request attempting to purge the read copy. This case is handled by the snooping on the RAC. When the RAC sees an invalidation request for a pending read, it changes the state of that RAC entry to invalidated-read-pending. In this state, the RC conservatively assumes that any read reply is stale and treats the reply as a NAK response.

4.7 Deadlock

In the DASH prototype, deadlocks are eliminated through a combination of hardware and protocol features. At the hardware level, DASH consists of two mesh networks, each of which guarantees point-to-point delivery of messages without deadlocks. However, this by itself is not sufficient to prevent deadlocks because the consumption of an incoming message may require the generation of another outgoing message. This can result in circular dependencies between the limited buffers present in two or more nodes and cause deadlock.

To address this problem, the protocol divides all messages into request messages (e.g. read and read-exclusive requests and invalidation requests) and reply messages (e.g. read and read-exclusive replies and invalidation acknowledges). Furthermore, one mesh is dedicated to servicing request messages while the other handles reply messages. Reply messages are guaranteed to be consumed at the destination, partly because of their nature and partly because space for the reply data is preallocated in the RAC. This eliminates the possibility of request-reply circular dependencies and the associated deadlocks.

However, the protocol also relies on request messages that generate additional requests. Because of the limited buffer space, this can result in deadlocks due to request-request circular dependencies. Fairly large input and output FIFO's reduce the probability of this problem. If it does arise, the directory hardware includes a time-out mechanism to break the possible deadlock. If the directory has been blocked for more than the time-out period in attempting to forward a request it will instead reject the request with a NAK reply message. Once this deadlock breaking mode is entered enough other requests are handled similarly so that any possible deadlock condition that has arisen within the request network can be eliminated. As in cases discussed earlier, this scheme relies on the processor's ability to reissue its request upon receiving a NAK.

4.8 Error Handling

The final set of exceptions arise in response to error conditions in the hardware or protocol. The system includes a number of error checks including ECC on main memory, parity on the directory memory, length checking of network messages and inconsistent bus and network message checking. These checks are reported to processors through bus errors and associated error capture registers. Network errors and improper requests are dropped by the receiver of such messages. Depending upon the type of network message that was lost or corrupted, the issuing processor will eventually time-out its originating request or some fence operation which will be blocked waiting for a RAC entry to be deallocated. The time-out generates a bus-error which interrupts the processor. The processes using the particular memory location are aborted, but low level operating system code can recover from the error if it is not within the kernel. The OS can subsequently clean up the state of a line by using back-door paths that allow direct addressing of the RAC and directory memory.

5 Supplemental Operations

During the evolution of the DASH protocol, several additional memory operations were evaluated. Some of these operations are included in the DASH prototype, while others were not included due to hardware constraints or a lack of evidence that the extension would provide significant performance gains.

The first major extension incorporated into the DASH protocol was support for synchronization operations. The sharing characteristics of synchronization objects are often quite different from those of normal data. Locks, barriers, and semaphores can be highly contended. Using the normal directory protocol for synchronization objects can lead to hot spots. For example, when a highly contended lock is released, all processor caches containing the lock are invalidated; this invalidation results in the waiting processors rushing to grab the lock. DASH provides special *queue-based lock* primitives that use the directory memory to keep track of clusters waiting for a lock. Using the directory memory is natural since it is already set up to track queued clusters, and the directory is normally accessed in read-modify-write cycles that match the atomic update necessary for locks. An unlock of a queue-based lock while clusters are waiting results in a grant of the lock being sent to one of the waiting clusters. This grant allows the cluster to obtain the lock without any further network messages. Thus, queue-based locks reduce the hot spotting generated by contended locks and reduce the latency between an unlock operation and subsequent acquisition of the lock. This and other synchronization primitives are discussed in detail in [17].

Another set of operations included in the prototype help hide the latency of memory operations. Normally, when a read is issued the processor is stalled until the data comes back. With very fast processors, this latency can be tens to hundreds of processor cycles. Support for some form of prefetch can clearly help. DASH supports both *read prefetch* and *read-exclusive prefetch* operations [17]. These operations cause the directory controller to send out a read or read-exclusive request for the data, but do not block the processor. Thus, the processor is able to overlap the fetching of the data with useful work. When the processor is ready to use the prefetched data, it issues a normal read or read exclusive request. By this time the data will either be in the RAC or the prefetch will be outstanding, in which case the normal read or read-exclusive is merged with the prefetch. In either case, the latency for the data will be reduced. Ideally, we would have liked to place the prefetched data directly in the requesting processor's cache instead of the RAC, but that would have required significant modifications to the existing processor boards.

There are some variables for which a write-update coherence protocol is more appropriate than the DASH write-invalidate protocol [3]. The prototype system provides for a single word *update write* primitive which updates memory and all the caches currently holding the word. Since exclusive ownership is not required, the producer's write buffer can retire the write as soon as it has been issued on the bus. Update-writes are especially useful for event synchronization. The producer of an event can directly update the value cached by the waiting processor reducing the latency and traffic that would result if the value was invalidated. This primitive is especially useful in implementing barriers, as an update-write can be used by the last processor entering the barrier to release all waiting processors. Update operations conform to the release consistency memory model, but require explicit fence operations when used for synchronization purposes.

6 Scalability of the DASH Directory

The DASH directory scheme currently uses a full bit-vector to identify the remote clusters caching a memory block. While this is reasonable for the DASH prototype, it does not scale well since the amount of directory memory required is the proportional to the product of the main memory size and the number of processors in the system. We are currently investigating a variety of solutions which limit the overhead of directory memory. The most straightforward modification is the use of a limited number of pointers per directory entry. Each directory pointer holds the cluster number of a cluster currently caching the given line. In any limited pointer scheme some mechanism must exist to handle cache blocks that are cached by more processors then there are pointers. A very simple scheme resorts to a broadcast in these cases [1]. Better results can be obtained if the pointer storage memory reverts to a bit vector when pointer overflow occurs. Of course, a complete bit vector is not possible, but if each bit represents a *region* of processors the amount of traffic generated by such overflows can be greatly reduced relative to a broadcast.

Other schemes to scale the directory rely on restructuring of directory storage. Possible solutions include allowing pointers to be shared between directory entries, or using a cache of directory entries to supplement or replace the normal directory [18, 13]. A directory structured as a cache need not have a complete backing memory since replaced directory entries can simply invalidate their associated cache entries (similar to how multi-level caches maintain their inclusion property). Recent studies [13] have shown that such *sparse-directories* can maintain a constant overhead of directory memory compared with a full-bit vector when the number of processors grows from 64 to 1024. A sparse directory using limited pointers and a coarse vector only increases the total traffic by only 10-20% and should have minimal impact on processor performance. Furthermore, such directory structures require only small changes to the coherence protocol given here.

7 Validation of the Protocol

Validation of the DASH protocol presents a major challenge. Each cluster in DASH contains a complex directory controller with a large amount of state. This state coupled with the distributed nature of the DASH protocol results in an enormous number of possible interactions between the controllers. Writing a test suite that exercises all possible interactions in reasonable time seems intractable. Therefore, we are using two less exhaustive testing methods. Both these methods rely on the software simulator of DASH that we have developed.

The simulator consists of two tightly coupled components: a low-level DASH system simulator that incorporates the coherence protocol, and simulates the processor caches, buses, and interconnection network at a very fine level of detail; and Tango [11], a high-level functional simulator that models the processors and executes parallel programs. Tango simulates parallel processing on a uniprocessor while the DASH simulator provides detailed timing about latency of memory references. Because of the tight coupling between the two parts, our simulator closely models the DASH machine.

Our first scheme for testing the protocol consists of running existing parallel programs for which the results are known and comparing the output with that from the DASH simulator. The drawback of using parallel programs to check the protocol is that they use the memory system and synchronization features in "well-behaved" ways. For example, a well-written parallel program will not release a lock that is already free, and parallel programs usually don't modify shared variables outside of a critical section. As a result, parallel applications do not test a large set of possible interactions.

To get at the more pathological interactions, our second method relies on test scripts. These scripts can be written to provide a fine level of control over the protocol transitions and to be particularly demanding of the protocol. While writing an exhaustive set of such test scripts is not feasible, we hope to achieve reasonable test coverage with a smaller set of scripts by introducing randomness into the execution of the scripts.

The randomness idea used is an extension of the Berkeley Random Case Generation (RCG) technique [22] used to verify the SPUR cache controller design. Our method, called Intelligent Case Generation (ICG), is described in detail in [14]. Each script is a self-contained test sequence which executes a number of memory operations on a set of processors. Each script consists of some initialization, a set of test operations, and a check for proper results. Like RCG, multiple, independent scripts run simultaneously and interact in two ways. First, a processor randomly chooses which of the multiple active scripts it is going to pick its next action from. Therefore, execution of the same set of scripts will be interleaved in time differently upon each run. Second, while each script uses unique memory locations, these locations may be in the same cache line. Scripts interact by changing the cache state of cache lines used by other scripts.

ICG extends RCG in three ways. First, instead of simple two step scripts (a write followed by a read), ICG supports multistep scripts in which some steps are executed in series and some are allowed to execute in parallel. Second, ICG provides a finer level of control over which processors execute which steps of a script and introduces randomness into the assignment process. Finally, ICG allows for a more flexible assignment of test addresses so that particular scripts do not have to be written to interact. Using ICG to dynamically assign addresses results in different scripts interacting at different times during a run, and results in the same script using various combinations of local and remote memory.

Of course, the hardware itself will also serve as a verification tool. The hardware can run both parallel programs and test scripts. While debugging protocol errors on the hardware will be difficult, the sheer number of cycles executed will be a demanding test of the protocol.

8 Comparison with Scalable Coherent Interface Protocol

Several protocols that provide for distributed directory-based cache coherence have been proposed [15, 21]. The majority of these protocols have not been defined in enough detail to do a reasonable comparison with the DASH protocol. One exception is the IEEE P1596 - Scalable Coherent Interface (SCI) [12]. While still evolving, SCI has been documented in sufficient detail to make a comparison possible. SCI differs from DASH, however, in that it is only an interface standard, not a complete system design. SCI only specifies the interfaces that each processor should implement, leaving open the actual node design and exact interconnection network.

At the system level, a typical SCI system would be similar to DASH with each processing node containing a processor, a section of main memory, and an interface to the interconnection network. Both systems rely on coherent caches maintained by distributed directories and distributed memories to provide scalable memory bandwidth. The major difference lies in how and where the directory information is maintained. In SCI, the directory is a distributed sharing list maintained by the processor caches themselves. For example, if processors A, B, and C are caching some location, then the cache entries storing this location will form a doubly-linked list. At main memory, only a pointer to the processor at the head of the linked list is maintained. In contrast, DASH places all the directory information with main memory.

The main advantage of the SCI scheme over DASH is that the amount of directory pointer storage grows naturally with the number of processors in the system. In DASH, the maximum number of processors must be fixed beforehand, or the system must support some form of limited directory information. On the other hand, the SCI directory memory would normally employ the same SRAM technology used by the processor caches while the DASH directory is implemented in main memory DRAM technology. Another feature of SCI is that it guarantees forward progress in all cases, including the pathological "live-lock" case alluded to in section 4.6.

The primary disadvantage of the SCI scheme is that the distribution of the individual directory entries increases the complexity and latency of the directory protocol, since additional directory update messages must be sent between processor caches. For example, on a write to a shared block cached by $N + 1$ processors (including the writing processor), the writer must perform the following actions: (i) detach itself from the sharing list; (ii) interrogate memory to determine the head of the sharing list; (iii) acquire head status from the current head; and (iv) serially purge the other processor caches by issuing invalidation requests and receiving replies indicating the next processor in the list. Altogether, this amounts to $2N + 8$ messages including N serial directory lookups. In contrast, DASH can locate all sharing processors in a single directory lookup and invalidation messages are serialized only by the network transmission rate. Likewise, many read misses in SCI require more inter-node communication. For example, if a block is currently cached, processing a read miss requires four messages since only the head can supply the cache block. Furthermore, if a miss is replacing a valid block in the processor's cache, the replaced block must be detached from its sharing list.

Recently, the SCI working committee has proposed a number of extensions to the base protocol that address some of these shortcomings. In particular, the committee has proposed additional directory pointers that allow sharing lists to become sharing trees, the support for request forwarding, and the use of a clean cached state. While these extensions reduce the differences between the two protocols, they also add complexity The fundamental question is what set of features leads to better performance at a given complexity level. As in the design of other hardware systems, this requires a careful balance between optimizing the performance of common operations without adding undue complexity for uncommon ones. The lack of good statistics on scalable shared memory machines, however, makes the identification of the common cases difficult. Thus, a complete comparison of the protocols is likely to require actual implementations of both designs and much more experience with this class of machines.

9 Summary and Status

Distributed directory-based coherence protocols such as the DASH protocol allow for the scalability of shared-memory multiprocessors with coherent caches. The cost of scalability is the added complexity of directory based schemes compared with existing snoopy, bus-based coherence protocols. The complexity arises primarily from the lack of a single serialization point within the system and the lack of atomic operations Additional complexity stems simply from the larger set of components that interact to execute the protocol and the deeper hierarchy within the memory system.

Minimizing memory latency is of paramount importance in scalable systems. Support for coherent caches is the first step in reducing latency, but the memory system must also be optimized towards this goal. The DASH protocol attempts to minimize latency through the use of the release consistency model, cache-to-cache transfers, a forwarding control strategy and special purpose operations such as prefetch and update write. Adding these latency reducing features must, of course, be traded off with the complexity needed to support them. All of the above features were added without a significant increase in the complexity of the hardware.

Verification of a complex distributed directory-based cache coherence protocol is a major challenge. We feel that verification through the use of test scripts and extensive random testing will provide an acceptable level of confidence. The design effort of the prototype is currently in the implementation phase. A functional simulator of the hardware is running as well as a gate level simulation of the directory card. We plan to have a 4 cluster, 16 processor system running during the summer of 1990. This prototype should serve as the ultimate verification of the design and provide a vehicle to fully evaluate the design concepts discussed in this paper.

10 Acknowledgments

The DASH project team is composed of a number of graduate students and faculty within the Computer System Laboratory at Stanford. Many related topics are being researched and the results of much of this work has influenced the design of the DASH architecture and coherence protocol. Besides the authors, a number of others have directly contributed to the development of DASH. In particular, we would like to thank Wolf-Dietrich Weber for creating the DASH simulator, Helen Davis and Stephen Goldschmidt for modifying their Tango simulator to interact with the DASH simulator, and Bruce Kleinman for developing the DASH protocol verifier. Likewise, we want to recognize research engineer Dave Nakahira who has made significant contributions to the design of the DASH hardware. We also wish to thank Valid Logic Systems who has donated the CAE software used to develop the DASH prototype.

This research is supported by DARPA contract N00014-87-K-0828. Dan Lenoski is supported by Tandem Computers Incorporated.

References

[1] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *Proc. of the 15th Annual Int. Sym. on Computer Architecture*, pages 280–289, June 1988.

[2] J. Archibald and J.-L. Baer. An economical solution to the cache coherence problem. In *Proc. of the 12th Int. Sym. on Computer Architecture*, pages 355–362, June 1985.

[3] J. Archibald and J.-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Trans. on Computer Systems*, 4(4):273–298, 1986.

[4] F. Baskett, T. Jermoluk, and D. Solomon. The 4D-MP graphics superworkstation: Computing + graphics = 40 MIPS + 40 MFLOPS and 100,000 lighted polygons per second. In *Proc. of the 33rd IEEE Computer Society Int. Conf. - COMPCON 88*, pages 468–471, February 1988.

[5] W. C. Brantley, K. P. McAuliffe, and J. Weiss. RP3 processor-memory element. In *Proc. of the 1985 Int. Conf. on Parallel Processing*, pages 782–789, 1985.

[6] L. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Trans. on Computers*, C-27(12):1112–1118, December 1978.

[7] W. J. Dally. Wire efficient VLSI multiprocessor communication networks. In *Stanford Conference on Advanced Research in VLSI*, 1987.

[8] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *Proc. of the 13th Annual Int. Sym. on Computer Architecture*, pages 434–442, June 1986.

[9] C. M. Flaig. VLSI mesh routing systems. Technical Report 5241:TR:87, California Institute of Technology, May 1987.

[10] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. of the 17th Annual Int. Sym. on Computer Architecture*, June 1990.

[11] S. R. Goldschmidt and H. Davis. Tango introduction and tutorial. Technical Report CSL-TR-90-410, Stanford University, January 1990.

[12] P1596 Working Group. P1596/Part IIIA - SCI Cache Coherence Overview. Technical Report Revision 0.33, IEEE Computer Society, November 1989.

[13] A. Gupta and W.-D. Weber. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. Technical Report CSL-TR-90-417, Stanford University, March 1990.

[14] B. Kleinman. *DASH Protocol Verification*, EE-391 Class Project Report, December 1989.

[15] T. Knight. Architectures for artificial intelligence. In *Int. Conf. on Computer Design*, 1987.

[16] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9):241–248, September 1979.

[17] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. Design of the Stanford DASH multiprocessor. Technical Report CSL-TR-89-403, Stanford University, December 1989.

[18] B. O'Krafka and A. R. Newton. An empirical evaluation of two memory-efficient directory methods. In *Proc. of the 17th Annual Int. Sym. on Computer Architecture*, June 1990.

[19] M. S. Papamarcos and J. H. Patel. A low overhead coherence solution for multiprocessors with private cache memories. In *Proc. of the 11th Annual Int. Sym. on Computer Architecture*, pages 348–354, June 1984.

[20] C. K. Tang. Cache design in the tightly coupled multiprocessor system. In *AFIPS Conf. Proc., National Computer Conf., NY, NY*, pages 749–753, June 1976.

[21] J. Willis. Cache coherence in systems with parallel communication channels & many processors. Technical Report TR-88-013, Philips Laboratories - Briarcliff, March 1988.

[22] D. A. Wood, G. A. Gibson, and R. H. Katz. Verifying a mulitprocessor cache controller using random case generation. Technical Report 89/490, University of California, Berkeley, 1988.

[23] W. C. Yen, D. W. Yen, and K.-S. Fu. Data coherence problem in a multicache system. *IEEE Trans. on Computers*, C-34(1):56–65, January 1985.

Appendix A: Coherence Transaction Details

```
if (Data held locally in shared state by processor or RAC)
   Other cache(s) supply data for fill;

else if (Data held locally in dirty state by processor or RAC) {
   Dirty cache supplies data for fill and goes to shared state;
   if (Memory Home is Local)
      Writeback Data to main memory;
   else
      RAC takes data in shared-dirty state;
   }

else if (Memory home is Local) {
   if (Directory entry state != Dirty-Remote)
      Memory supplies read data;
   else {
      Allocate RAC entry, mask arbitration and force retry,
      Forward Read Request to Dirty Cluster;
      PCPU on Dirty Cluster issues read request.
      Dirty cache supplies data and goes to shared state;
      DC sends shared data reply to local cluster;
      Local RC gets reply and unmasks processor arbitration;
      Upon local processor read, RC supplies data and the
         RAC entry goes to shared state.
      Directory entry state = Shared-Remote;
      }
   }

else /* Memory home is Remote */ {
   Allocate RAC entry, mask arbitration and force retry.
   Local DC sends read request to home cluster;
   if (Directory entry state != Dirty-Remote) {
      Directory entry state = Shared-Remote, update vector;
      Home DC sends reply to local RC;
      Local RC gets reply and unmasks processor Arbitration;
   else {
      Home DC forwards Read Request to dirty cluster,
      PCPU on dirty cluster issues read request and DC sends
         reply to local cluster and sharing writeback to home;
      Local RC gets reply and unmasks processor arbitration;
      Home DC gets sharing writeback, writes back dirty data,
         Directory entry state = Shared-Remote, update vector;
      }
   Upon local processor read, RC supplies the data and the
      RAC entry goes to shared state;
   }
```

Figure 7: Normal flow of read request bus transaction.

```
if (Memory Home is Local) {
   Writeback data is written back into main memory.
   }
else /* Memory Home is Remote */ {
   Writeback request sent to home,
   Writeback data is written back into main memory.
   Directory entry state = Uncached-Remote, update vector,
   }
```

Figure 8: Normal flow of a write-back request bus transaction.

```
if (Data held locally in dirty state by processor or RAC)
   Dirty cache supplies Read-Exclusive fill data and
      invalidates self,

else if (Memory Home is Local)  {
   switch (Directory entry state) {

      case Uncached-Remote
         Memory supplies data, any locally cached copies
            are invalidated.
         break;

      case Shared-Remote .
         RC allocates an entry in RAC with DC specified
            invalidate acknowledge count.
         Memory supplies data, any locally cached copies are
            invalidated;
         Local DC sends invalidate request to shared clusters.
         Dir  entry state = Uncached Remote, update vector,
         Upon receipt of all acknowledges RC deallocates RAC
            entry;
         break;

      case Dirty-Remote
         Allocate RAC entry, mask arbitration and force retry.
         Forward Read-Exclusive Request to dirty cluster.
         PCPU at dirty cluster issues Read-Ex request,
            Dirty cache supplies data and invalidates self,
         DC in dirty cluster sends reply to local RC,
         Local RC gets reply from dirty cluster and unmasks
            processor arbitration,
         Upon local processor re-Read-Ex, RC supplies data,
            RAC entry is deallocated and
            Dir. entry state = Uncached-Remote, update vector,
      }
   }
else /* Memory Home is Remote */ {
   RC allocates RAC entry, masks arbitration and forces retry;
   Local DC sends Read-Exclusive request to home;
   switch (Directory entry state) {

      case Uncached-Remote .
         Home memory supplies data, any locally cached copies
            are invalidated, Home DC sends reply to local RC.
         Directory entry state = Dirty-Remote, update vector,
         Local RC gets Read-Ex reply with zero invalidation
            count and unmasks processor for arbitration,
         Upon local processor re-Read-Ex, RC supplies data and
            RAC entry is deallocated;
         break;

      case Shared-Remote ·
         Home memory supplies data, any locally cached copies
            are invalidated, Home DC sends reply to local RC,
         Home DC sends invalidation requests to sharing
            clusters;
         Directory entry state = Dirty-Remote, update vector,
         Local RC gets reply with data and invalidate acknow-
            ledge count and unmasks processor for arbitration.
         Upon local processor re-Read-Ex, RC supplies data.
         Upon receipt of all acknowledges RC deallocates RAC
            entry;
         break;

      case Dirty-Remote
         Home DC forwards Read-Ex request to dirty cluster.
         PCPU at dirty cluster issues Read-Ex request,
            Dirty cache supplies data and invalidates self.
         DC in dirty cluster sends reply to local RC with
            acknowledge count of one and sends Dirty Transfer
            request to home;
         Local RC gets reply and acknowledge count and unmasks
            processor for arbitration.
         Upon local processor re-Read-Ex. RC supplies data.
         Upon receipt of Dirty Transfer request, Home DC
            sends acknowledgment to local RC,
          Home Dir  entry state = Dirty-Remote, update vector,
         Upon receipt of acknowledge RC deallocates RAC entry.
      }
   }
```

Figure 9: Normal flow of read-exclusive request bus transaction.

An Algorithmic Approach to Compound Loop Transformation

Michael E. Wolf and Monica S. Lam
Computer Systems Laboratory
Stanford University, CA 94305

Abstract

This paper presents a theory that unifies many existing loop transformations, including loop interchange or permutation, skewing, reversal, tiling, and combinations of these elementary transformations. This theory provides the foundation for solving an open question in compilation for parallel machines: Which loop transformations and, in what order, should be applied to achieve a particular goal, such as maximizing parallelism or data locality. This paper presents an efficient loop transformation algorithm based on this theory to maximize the degree of parallelism in a loop nest.

1 Introduction

Loop transformations, such as loop interchange, reversal, skewing and tiling (or subblocking)[2, 4, 18] have been shown to be useful for two important goals: parallelism and efficient use of the memory hierarchy. Previous work on loop transformations focused on the application of *individual* transformations: when it is legal to apply a transformation, and if the transformation directly contributes to a particular goal. It remains an open question as to how to combine these transformations to optimize general loop nests for a particular goal. This paper introduces a theory of loop transformations that answers this question.

A technique commonly used in today's parallelizing compilers is to decide *a priori* the order in which the compiler should attempt to apply transformations. This technique is inadequate because the effectiveness of a given transformation often depends on the future transformations that can be applied. Another proposed technique is to "generate and test", that is, to apply all different possible combinations of transformations. This "generate and test" approach is expensive. Differently transformed versions of the same program may trivially have the same behavior and so need not be explored. For example, when vectorizing, the order of the outer loops is not significant. More importantly, generate and test approaches cannot search the entire space of transformations that have potentially infinite instantiations. Loop skewing is such a transformation, since a wavefront can travel in an infinite number of different directions.

For loops whose data dependences are distance vectors, a more rigorous mathematical approach has been proposed. In this approach, loop interchange, reversal, and skewing transformations are unified as linear transformations in the iteration space. This mathematical formulation of a loop has been used in the study of generating systolic arrays and tiling[6, 7, 10, 11, 14, 15]. The restriction that data dependences must be distance vectors excludes loops that contain any "serializing" loops. That is, in this notation,

This research was supported in part by DARPA contract N00014-87-K-0828.

an n-dimensional iteration space trivially can be transformed to produce $n - 1$ parallel loops. We are interested in representing general loop nests and transforming the loops to maximize the number of parallel loops.

Our approach combines the rigor of the iteration space approach with the general program domain of the vectorizing and concurrentizing compilers. Our dependence vectors can incorporate both distance and direction information. The various transformations, interchange, reversal and skewing are unified as linear transformations. Compound transformations are just another linear transformation. This unification provides a general condition to determine if the code obtained via a compound transformation is legal, as opposed to a specific test for each individual elementary transformation. This makes it possible to search through the transformation space efficiently to achieve a given goal. Moreover, the relationships and interactions between different transformations can be analyzed in this unified model. Similarly, this model supports the derivation of the new loop bounds directly after a compound transformation. If loop bounds were derived for every transformation, the final expressions derived may be more complex than necessary.

Using this notation, we have developed algorithms for improving the parallelism and locality of a loop nest via loop transformations. Our parallelizing algorithm maximizes the degree of parallelism, that is, the number of parallel loops, within a loop nest. By finding the maximum number of parallel loops, multiple consecutive loops can be coalesced to form a single loop with all the iterations; this facilitates load balancing and reduces synchronization overhead. The different degrees of parallelism can be exploited directly by processors with different levels of parallelism, such as a multiprocessor with superscalar nodes. Moreover, some of the loops may contain a small number of loop iterations. Parallelizing only one loop may not fully exploit all the parallelism in the machine. The algorithm can generate coarse-grain and/or fine-grain parallelism; the former is useful in multiprocessor organizations and the latter is useful for vector machines and superscalar machines, machines that can execute multiple instructions per cycle. It can generate code for machines that can use multiple levels of parallelism, such as a multiprocessor with vector nodes.

We have also applied our representation of transformations successfully to the problem of data locality. All modern machine organizations, including uniprocessors, employ a memory hierarchy to speed up data accesses; the memory hierarchy typically consists of registers, caches, primary memory and secondary memory. To use this memory hierarchy efficiently, our locality optimization seeks to maximize the reuse of data that has been recently accessed. As the processor speed improves and the gap between processor and memory speeds widens, data locality becomes more important. Even with very simple machine models (for example, uniprocessors with data caches), complex compound loop transformations may be necessary [8, 9, 13]. The consideration of data locality makes it more important to be able to combine primitive loop transformations in a systematic manner.

The loop transformation algorithm has been implemented in our Stanford University parallelizing compiler. The implementation has taken only about two man-months, demonstrating that the implementation is made simple by the theory.

This paper introduces our model of loop dependences and transformations. We describe how the model facilitates the application of compound transformation, using parallelism as our target. The model is important in that it enables the choice of an optimal transformation without an exhaustive search. Here we will only present the parallelization algorithm; the proof that it finds the optimal transformation[16] is outside the scope of this paper. The derivation of the optimal compound transformation consists of two steps. The first step puts the loops into a canonical form, and the second step tailors it to specific architectures. While the first step can be expensive in the worst case, we have developed an algorithm that is feasible in practice. We apply a cheaper technique to handle as many loops as possible, and use the more general and expensive technique only on the remaining loops. We expect to find the optimal transformation in $O(n^3d)$ time for most programs, where n is the depth of the loop nests and d is the number of dependence vectors. The second step of specializing the code for different granularities of parallelism is straightforward and cheap. After deciding on the compound transformation to apply, the code including the loop bounds is then modified.

The organization of the paper is to first present the representation of the loop nests and the modeling of loop transformations. After establishing the notation, we illustrate the algorithm of parallelization by stepping through a simple example and showing the output code for different machine organizations. Finally, we describe a method for deriving the bounds of a loop after a compound transformation.

2 Representation

2.1 Program Representation

Our approach is applicable to perfectly nested loop nests; we assume that all optimizations have been applied to create perfectly loop nests whenever possible [1]. The upper and lower loop bounds must be linear expressions of the loop indices and the loops are normalized to have unit step sizes. In our model, a loop nest of depth n corresponds to a finite convex polyhedron of iteration space $\mathcal{Z}^n$, bounded by the loop bounds. Each iteration in the loop corresponds to a *node* in the polyhedron, and is identified by its index vector $\vec{p} = (p_1, p_2, \ldots, p_n)$; p_i is the loop index of the i loop in the nest, counting from the outermost to innermost loop. In the sequential program, the iterations are therefore executed in lexicographic order of their index vectors.

The scheduling constraints of the loop are represented as dependence vectors. The only dependences of interest are loop carried dependences, and not loop independent dependences. It is not necessary to classify the different dependence types such as control, anti- or output dependence, nor is the identity of the related memory accesses of any significance.

A dependence vector in an n-nested loop is denoted by a vector $\vec{d} = (d_1, d_2, \ldots, d_n)$. Each component d_i is a range of integers, represented by

$$[d_i^\mu, d_i^\nu], \text{where } d^\mu \in \mathcal{Z} \cup \{-\infty\}, d^\nu \in \mathcal{Z} \cup \{\infty\} \text{ and } d^\mu \le d^\nu.$$

A single dependence vector represents a set of distance vectors, known as the *distance vector set*:

$$\mathcal{E}(\vec{d}) = \{(e_1, \ldots, e_n) | e_i \in \mathcal{Z} \wedge d_i^\mu \le e_i \le d_i^\nu\}.$$

Each distance vector defines a set of edges on pairs of nodes in the iteration space. We say that an edge $(\vec{p_1}, \vec{p_2})$ exists if and only if $\exists \vec{e} \in \mathcal{E}(\vec{d})$ for some dependence vector $\vec{d}$, such that $\vec{p_2} = \vec{p_1} + \vec{e}$. The dependence vectors thus define a partial order on the nodes in the iteration space, and any topological ordering on the graph is a legal execution order, as all dependences in the loop are satisfied.

This notation allows us to represent both direction [3, 17] and distance information in a uniform notation. For example, the Wolfe direction vector '$<$' would be represented in our notation as $d^\mu = 1$ and $d^\nu = \infty$, or $[1, \infty]$ for short. If a dependence has a constant distance δ, then the dependence is represented as $[\delta, \delta]$, and we use the shorthand δ to represent that distance when the context is clear.

Finite ranges of distance components are represented by separate dependence vectors; that is, if $d^\mu \neq d^\nu$, then $d^\mu = -\infty$ or $d^\nu = \infty$ or both.

The arithmetic and comparison operators over the domain of components are defined in a straightforward way to give useful meanings. For example, arithmetic operators are defined so that $2 + [-3, \infty] = [-1, \infty]$. We also utilize the multiplication of a distance by a scalar when taking dot products. We use the straightforward definition that

$$s[a,b] = \begin{cases} [sa, sb], & \text{if } s \geq 0 \\ [sb, sa], & \text{otherwise} \end{cases}$$

and $s \cdot \infty$ is ∞ for positive s, 0 if s is 0, and $-\infty$ for negative s, and likewise for a factor times $-\infty$. These definitions of addition and multiplication are conservative in that

$$\vec{e}_1 \in \mathcal{E}(\vec{d}_1) \text{ and } \vec{e}_2 \in \mathcal{E}(\vec{d}_2) \Rightarrow f(\vec{e}_1, \vec{e}_2) \in \mathcal{E}(f(\vec{d}_1, \vec{d}_2))$$

where f is a function that performs a combination of multiplications and additions on its operands. The converse, that

$$\vec{e} \in \mathcal{E}(f(\vec{d}_1, \vec{d}_2)) \Rightarrow \exists \vec{e}_1 \in \mathcal{E}(\vec{d}_1) \wedge \vec{e}_2 \in \mathcal{E}(\vec{d}_2) : f(\vec{e}_1, \vec{e}_2) = \vec{e},$$

is not necessarily true unless $\vec{d}_1$ and $\vec{d}_2$ are themselves distance vectors.

A component d is *positive*, written $d > 0$, if its minimum d^μ is a positive integer. It is *non-negative*, written $d \geq 0$, if its minimum is non-negative. Likewise, d is *negative* or *non-positive* if its maximum d^ν is negative or non-positive respectively. We use the notation '$+$' as shorthand for $[1, \infty]$, '$-$' as shorthand for $[-\infty, -1]$, and '$*$' as shorthand for $[-\infty, \infty]$.

Since the nodes are initially executed in lexicographic order, the scheduling constraints can be captured by a set of *lexicographically positive* dependence vectors. A dependence vector $\vec{d}$ is lexicographically positive, written $\vec{d} \succ \vec{0}$, if $\exists i : (d_i > 0 \text{ and } \forall j < i : d_j \geq 0)$. A dependence vector $\vec{d}$ is lexicographically non-negative, written $\vec{d} \succeq \vec{0}$, if it is lexicographically positive or its components are all non-negative. A zero vector is one with all components equal to 0, written $\vec{0}$.

The procedure for extracting data dependence for this representation is similar to those used in previous vectorizing and parallelizing compilers. The only difference is that we require the data dependences of the original programs be represented as lexicographically positive data dependence vectors. For example:

```
for $i := 0$ to $n$ do
  for $j := 0$ to $n$ do
    $b := g(b);$
```

The dependence vectors are $\{(0, \text{'+'}), (\text{'+'}, \text{'*'})\}$. The lexicographical positive property of the dependences is crucial in simplifying the modeling of loop transformations.

2.2 Transformation Representation

The scope of loop transformations addressed in this paper is restricted to transformations that manipulate entire iterations and reorganize them within a loop nest. A loop transformation is defined by two mapping functions. The first is a one-to-one and onto mapping between a node in the convex polyhedron representing the original loop nest and a node in another convex polyhedron in an iteration space of possibly different dimensions. The second function maps the original set of dependence vectors, such that if a distance vector exists between a pair of nodes in the original loop, one also exists between the corresponding nodes in the transformed loop. We note that there may not be a one-to-one correspondence between the dependence vectors of the two loops because infinite sets of distance vectors can be represented only along the basis of the iteration space. This notation is chosen because it is efficient and it captures most of the dependences found in real programs. We say that a transformation is *valid* if the transformed dependence edges are acyclic. Traditionally, we say that it is *legal* to apply a transformation to a loop nest if the transformed code can be executed sequentially, or in lexicographic order of the iteration space. We observe that if nodes in the transformed code are executed in lexicographic order, all data dependences are satisfied if the transformed dependence vectors are lexicographically positive. This observation leads to a general definition of a legal transformation.

Definition 2.1 *A transformation is* legal *if the transformed dependence vectors are all lexicographically positive.*

Many of the loop transformations used in vectorizing and parallelizing compilers can be generalized as *linear transformations*; these include permutation, reversal and skewing. An important non-linear loop transformation is tiling.

2.2.1 Linear Transformations

A linear transformation T, where T is a non-singular, unimodular matrix, maps iteration $\vec{p}$ to iteration $T\vec{p}$ and dependence vector $\vec{d}$ to iteration $T\vec{d}$. T is unimodular so that T^{-1} maps the transformed iteration $\vec{p'}$ back to integral points in the original iteration space $\vec{p} = T^{-1}\vec{p'}$. We consider only $n \times n$ matrices, where n is the nest depth. Three of the common loop transformations, permutation, reversal and skewing, are elementary transformations.

- *Permutation:* A permutation σ on a loop nest transforms iteration $(p_1, \ldots, p_n)$ to $(p_{\sigma_1}, \ldots, p_{\sigma_n})$. This transformation can be expressed in matrix form as I_σ, the $n \times n$ identity matrix I with rows permuted by σ.

- *Reversal:* Reversal of loop i is represented by the identity matrix, but with the ith diagonal element equal to -1 rather than 1.

- *Skewing:* Skewing loop l_j by an integer factor f with respect to loop l_i [18] maps iteration

  $$(p_1, \ldots, p_{i-1}, p_i, p_{i+1}, \ldots, p_{j-1}, p_j, p_{j+1}, \ldots, p_n)$$

  to

  $$(p_1, \ldots, p_{i-1}, p_i, p_{i+1}, \ldots, p_{j-1}, p_j + fp_i, p_{j+1}, \ldots, p_n).$$

  The transformation matrix that produces skewing is the identity matrix, but with the element $t_{i,j}$ equal to f rather than zero. Since $i < j$, T must be lower triangular.

A compound transformation can be synthesized from a sequence of primitives, and the effect of the transformation is represented by the products of the various transformation matrices for each primitive transformation. Such a transformation is always unimodular. If the computation is to be executed sequentially in lexicographic order, then it must be the case that $T\vec{d} \succ \vec{0}$. This observation allows us to devise a simple legality test for general linear transformations.

Theorem 2.1 (Linear Transformation Test). *Let D be the set of dependence vectors of a computation. A linear transformation T is legal if T is non-singular and unimodular, and if $\forall \vec{d} \in D : T\vec{d} \succ \vec{0}$.*

The proof is a simple consequence of the definition of legal and that if $\vec{d} \in D$ then $\vec{e} \in \mathcal{E}(\vec{d}) \rightarrow T\vec{e} \succ \vec{0}$. Since our arithmetic operators are only conservative for general dependence vectors, it is the case that $\vec{d} \in D$ then $T^{-1}(T\vec{d}) = \vec{d}$ only under the two common cases stated in Theorem 2.2.

Theorem 2.2 *Let D be the set of dependence vectors of a computation. Suppose either of the following is true:*

1. *all $\vec{d} \in D$ are distance vectors, or*
2. *the linear transformation T can be synthesized exclusively from a combination of permutation and loop reversals.*

Then the linear transformation T is legal if and only if $\forall \vec{d} \in D : T\vec{d} \succ \vec{0}$.

As an example, let us consider the following code:

```
for $i$ := 1 to $n$ do
  for $j$ := 1 to $n$ do
    $a[i,j] := f(a[i,j], a[i+1, j-1])$;
  end for
end for
```

This code has the dependence $(1, -1)$. Loop interchange is represented by $T = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ mapping iteration (i, j) to (j, i). However, $T(1, -1)$ is $(-1, 1)$ which is lexicographically negative, rendering loop interchange illegal on this loop. On the other hand, the transformation represented by $T' = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$ is legal. Note that $\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ so the legal transformation can be considered an interchange followed by a reversal of the outermost loop.

We say that a set of adjacent loops i through j is *fully permutable* if it is legal to reorder the loops in all possible permutations. Full permutability is an important property that is exploited by transformations for both parallelism and data locality.

Theorem 2.3 (Full Permutability Test.) *Loops i through j of a legal computation with dependence vectors D are fully permutable if*

$$\forall d \in D : ((d_1, \ldots, d_{i-1}) \succ \vec{0} \vee (\forall i \leq k \leq j : d_k \geq 0))$$

2.2.2 Tiling

Tiling [17] is not a linear transformation. When we tile loops $i, \ldots, j$ by sizes $b_i, \ldots, b_j$, the iteration space gains $j - i + 1$ new dimensions, and the iteration

$$(p_1, \ldots, p_{i-1}, p_i, \ldots, p_j, p_{j+1}, \ldots, p_n)$$

is mapped to

$$(p_1, \ldots, p_{i-1}, p'_i, \ldots, p'_j, p''_i, \ldots, p''_j, p_{j+1} \ldots, p_n)$$

where $p'_k = \lfloor p_k / b_k \rfloor$, and $p''_k = p_k \bmod b_k$, for each $i \le k \le j$.

Define the function

$$s(d) = \begin{cases} 0 & \text{if } d = 0 \\ \text{'+'} & \text{if } d \neq 0 \text{ and } d \text{ non-negative} \\ \text{'-'} & \text{if } d \neq 0 \text{ and } d \text{ non-positive} \\ \text{'*'} & \text{otherwise.} \end{cases}$$

A dependence vector $(d_1, \ldots, d_n)$ is transformed into up to 2^{j-i+1} new vectors of the form

$$(d_1, \ldots, d_{i-1}, d'_i, \ldots, d'_j, d''_i, \ldots, d''_j, d_{j+1}, \ldots, d_n),$$

where for each $i \le k \le j$, either $d'_k = s(d_k)$ and $d''_k =$ '$*$' or $d'_k = 0$ and $d''_k = d_k$, except that if $d_k = 0$ then $d'_k = 0$ and $d''_k = 0$.

From examination of the above, it is clear that if loops i through j of a legal computation are fully permutable, then they are also tilable. Since the resulting dependence vectors are independent of the size of the tile, it is not necessary to determine that size at loop transformation time.

3 The Parallelizing Algorithm

Iterations of a loop can execute in parallel if and only if there are no dependences carried by that loop Suppose the loop nest $(p_1, \ldots, p_n)$ can be executed correctly in lexicographic order. The loop p_i of a legal sequential loop nest is parallelizable if and only if for all dependence vectors $(d_1, \ldots, d_{i-1}) \succ \vec{0}$ or $d_i = 0$. Such a loop is called a DOALL loop. To maximize the degree of parallelism is to transform the loop nest to maximize the number of loops that satisfy this property.

We divide the problem of parallelization into two parts. We first transform the original loop nest into nests of largest fully permutable loop nests. This is the canonical form from which maximum degrees of coarse and fine grain parallelism can be obtained. Then different techniques are applied to obtain the granularities of parallelism appropriate for the target machine. We illustrate this algorithm using the following example:

```
for i := 1 to n do
  for j := 1 to n do
    for k := 1 to n do
      (a[i,k],b[i,j,k]) := f(a[i,k], a[i+1,k-1], b[i,j,k], b[i,j,k-1]);
```

The loop body above is represented by a cube in a three-dimensional iteration space with sides of length n. Discarding $(0,0,0)$, the dependence vectors are,

$$D = \{(0, \text{'+'}, 0), (1, \text{'*'}, -1), (0, 0, 1)\}.$$

None of the three loops in the source program can be parallelized as it stands; however, there is one degree of parallelism that can be exploited at either a coarse or fine grain level.

In the description below, we will show the code resulted from each step of the transformation process. In reality, code is generated once only at the end of the entire algorithm.

3.1 Canonical Form

A loop is in canonical form for parallelization if it contains the maximally outermost fully permutable loops under linear transformations. Once in canonical form, the loops can be translated mechanically to suit a particular parallel architecture.

For the example above, the algorithm *permutes* the j and k loops, and *skews* the k loop with respect to loop i by a factor of 1, resulting in the code below:

```
for i := 1 to n do
  for k := i+1 to i+n do
    for j := 1 to n do
      (a[i,k-i],b[i,j,k-i]) := f(a[i,k-i], a[i+1,k-i-1], b[i,j,k-i], b[i,j,k-i-1]);
```

The transformation matrix T and the transformed dependences D are

$$T = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

and

$$D = \{(0,0,\text{`}+\text{'}),(1,0,\text{`}*\text{'}),(0,1,0)\}.$$

The transformation is legal since the dependences remain lexicographically positive. The first two loops form one set of fully permutable loop nest, since interchanging loops i and j leaves the dependences lexicographically positive. The loop k is in a (degenerate) set of permutable loops by itself.

We will briefly outline the algorithm that is used in transforming the code into canonical form[16]. The algorithm constructs the final set of loops incrementally starting with the outermost subnest and working inwards. The same procedure of finding the currently outermost, largest fully permutable loop nest is applied recursively. For each subnest, the algorithm adds loops to it one at a time. A loop may first be reversed and/or skewed with respect to outer loops before it can be permuted to be included into the current subnest. This permute-reverse-skew does not always deliver the optimal result. However, it is optimal in common cases such as when the nest contains less than four loops, or when all the dependences in the original program are distance vectors. In those cases where this algorithm cannot order all the loop, we apply general 2-D transformations[14, 15] on pairs of loops to improve parallelism. With this transformation, our algorithm is optimal for loops nests of depth four or less in $O(n^3d)$ where n is the loop nest depth and d is the number of dependences[16].

3.2 Targeting for Specific Architectures

In the following, we first show that the loops in the canonical format can be trivially transformed to give coarsest granularity of parallelism. We then show these loops can be transformed to give the same degree of fine-grain parallelism, suitable for superscalar and VLIW architectures. We then return to the multiprocessor architecture, and show how the same degree of parallelism can be obtained via lower synchronization cost, and how both fine- and coarse-grain parallelism can be produced.

3.2.1 Coarse Grain Parallelism

A nest of n fully permutable loops can be transformed to code containing at least $n - 1$ degrees of parallelism [11]. In the degenerate case when no dependences are carried by these n loops, the degree of parallelism is n. Otherwise, $n - 1$ loops can be obtained by skewing the innermost loop in the fully permutable nest by each of the other loops and moving the innermost loop to the outermost position. For example, the two-loop fully permutable set in the example above can be transformed to provide one level of parallelism:

```
for k := 3 to 3*n do
  doall i := max(1,⌈(k-n)/2⌉) to min(n,⌊(k-1)/2⌋) do
    for j := 1 to n do
      (a[i,k-2*i],b[i,j,k-2*i]) := f(a[i,k-2*i], a[i+1,k-2*i-1], b[i,j,k-2*i], b[i,j,k-2*i-1]);
```

The transformation matrix T for this phase of transformation, and the transformed dependences D are

$$T = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

and

$$D = \{(0,0,\text{'}+\text{'}),(1,1,\text{'}*\text{'}),(1,0,0)\}.$$

Applying this skew and interchange transformation to all the fully permutable loop nests will produce a loop nest with the maximum degree of parallelism. Moreover, the parallelism is contained in the outermost possible loops, and thus of the coarsest granularity possible[16].

3.2.2 Fine Grain Parallelism

If the target is a superscalar or VLIW machine, it is desirable that the parallel loop be innermost. If loop m is a parallel loop and $m < n$, then loop i can be permuted into the innermost loop via the transformation I_σ, where $\sigma = 1, \ldots, m-1, m+1, \ldots, n, m$. It is obvious that originally lexicographically positive dependences remain so if loop m is a parallel loop. Thus if there is a DOALL anywhere in the loop nest, we can create fine-grain parallelism for a machine that can use it. In fact, any number of DOALL loops can be permuted to be inner loops. This may be useful if code scheduling techniques such as software pipelining[12] are used. The overhead of starting and finishing a parallel loop is further reduced by coalescing the multiple DOALL loops. Since the transformation in Section 3.2.1 creates the largest possible number of DOALL loops, the maximum degree of fine-grain parallelism can be obtained by simply moving these DOALL loops innermost.

3.2.3 Reducing Global Barriers

Whenever a DOALL loop is nested within a non-DOALL loop, all processors must be synchronized at the end of each DOALL loop with a barrier. We can reduce the synchronization cost by tiling [17]. In the following, we show two variations.

After transforming the code to obtain the outermost fully-permutable loop nests, we do not skew and permute as suggested in Section 3.2.1. Instead, starting with the canonical form of the nest from Section 3.1, we tile the outermost fully-permutable nest. Using our simple example again, the tiled code becomes:

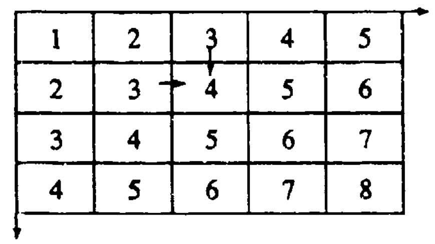

Figure 1: Order of DOALLs in tiled 2-dimensional iteration space

```
for ii := 1 to n by B do
   for kk := ii+1 to ii+n by B do
      for i := ii to min(ii+B, n) do
         for k := max(kk,i+1) to min(kk+B, n) do
            for j := 1 to n do
               (a[i,k-i],b[i,j,k-i]) := f(a[i,k-i], a[i+1,k-i-1], b[i,j,k-i], b[i,j,k-i-1]);
```

The outer loop nests obtained by tiling (*ii* and *kk* in the example) can be skewed and permuted to run in parallel just as the original loops. The advantage is that the synchronization cost is reduced by the block size. The i and k dimensions are plotted in Figure 1. Tiles are numbered by the index of their outer loops so that tiles with the same number are executed in parallel. Tiles numbered n cannot execute until all those numbered $n - 1$ have executed.

Tiling has two other advantages. First, within each tile, fine-grain parallelism can easily be obtained by skewing the loops *within* the tile and moving the DOALL loop innermost. Second, tiling can improve data locality if there is data reuse across several loops [8].

To further reduce the synchronization cost, we can apply the concept of a DOACROSS loop to the tile level [5] [17]. After tiling, instead of skewing the loops statically to form DOALL loops, the computation is allowed to skew dynamically by explicit synchronization between data dependent tiles. In the DOALL loop approach , tiles of each level must be completed before the processors may go on to the next, requiring a global barrier synchronization. In the DOACROSS model, each tile can potentially execute as soon as it is legal to do so. That is, referring to Figure 1, those numbered n can execute as soon as their neighbors that are numbered $n - 1$ have executed. This ordering can be enforced by local synchronization. Furthermore, different parts of the wavefront may proceed at different rates as determined dynamically by the execution times of the different tiles. In contrast, the machine must wait for the slowest processor at every level with the DOALL method.

3.3 Summary

We have outlined our two-step algorithm in finding parallelism for the different machines. The first is to transform the code into nests of maximal fully permutable loop nests. The second is to tailor the code to specific architectures. The step of transforming the loop nests into nests of fully permutable loops can be quite expensive, whereas the transformation of targeting to different machine architectures is straightforward.

4 Determining Loop Bounds

In this section we present a method to determine the loop bounds after a series of skews, permutations, reversals, general two dimensional transformations and tilings.

4.1 Scope

The class of loops that the loop bound calculation can handle is of the form

```
for $I_i := \max(L_i^1, L_i^2, \ldots)$ to $\min(U_i^1, U_i^2, \ldots)$ do
```

where

$$L_i^\alpha = \left\lceil \left(l_{i,0}^\alpha + \sum_{k=1}^{i-1} l_{i,k}^\alpha I_k \right) / l_{i,i}^\alpha \right\rceil$$

$$U_i^\alpha = \left\lfloor \left(u_{i,i}^\alpha + \sum_{k=1}^{i-1} u_{i,k}^\alpha I_k \right) / l_{i,i}^\alpha \right\rfloor$$

and all $l_{i,k}^\alpha$ and $u_{i,k}^\alpha$ are known constants, except possibly for $l_{i,0}^\alpha$ and $u_{i,0}^\alpha$, which must still be invariant in the loop nest. (If a ceiling occurs where we need a floor it is a simple matter to adjust $l_{i,0}^\alpha$ and $u_{i,0}^\alpha$ and replace the ceiling with the floor, and likewise if a floor occurs where we need a ceiling.) If any loop increments are not one, then they must first be made so, for example via loop normalization. If the bounds are not of the proper form, then the given loop cannot be involved in any transformations, and the loop nest is effectively divided into two: those outside the loop and those nested in the loop.

Loop skewing followed by permutation can easily produce bounds of this complexity, with minima, maxima, floors and ceilings. Since we wish to be able to take permuted and skewed loops and perform further transformations, we need full generality.

4.2 Determining the Bounds after Skewing or Reversal

Loop reversal can be implemented by negating the step and exchanging the upper and lower bounds, and applying loop normalization to make the step again unity. It is also easy to determine the bounds after loop skewing [18]. Moreover, if the bounds were previously in the class of bounds we can transform, then they remain so after the loop bound transformations for skewing and reversal.

4.3 Determining the Bounds after Permutation

We outline our method for determining the bounds of a loop nest after permutation by σ. We explain the general method and demonstrate it by permuting the following loop nest to make k the outermost loop and i the innermost loop.

```
for $i := 1$ to $n_i$ do
  for $j := 2i$ to $n_j$ do
    for $k := 2i + j - 1$ to $\min(j, n_k)$ do
      $S$;
```

The inequalities extracted from the above loop nests are:

$$\begin{array}{lll} i \geq 1 & i \leq n_i & \\ j \geq 2i & j \leq n_j & \\ k \geq 2i + j - 1 & k \leq j & k \leq n_k \end{array}$$

From these inequalities, we can find the maximum and minimum possible values of each loop index. This can be easily done by substituting the values obtained from the outermost loop to innermost. The minimum and maximum values are:

$$\begin{array}{ll} i \geq 1 & i \leq n_i \\ j \geq 2 \times 1 = 2 & j \leq n_j \\ k \geq 2 \times 1 + 2 - 1 = 3 & k \leq \min(n_j, n_k). \end{array}$$

We define for loop i

$$\min(I_i) = \max_{\alpha} (L_i^{*\alpha})$$

where

$$L_i^{*\alpha} = \left\lceil \left(l_{i,0}^{\alpha} + \sum_{k=1}^{i-1} l_{i,k}^{\alpha} I_{i,k}^{*\alpha} \right) / l_{i,i}^{\alpha} \right\rceil$$

and

$$I_{i,k}^{*\alpha} = \begin{cases} \min(I_k), & \operatorname{sgn}(l_{i,k}^{\alpha}) = \operatorname{sgn}(l_{i,k}^{\alpha}) \\ \max(I_k), & \text{otherwise.} \end{cases}$$

Similar formulas hold for $\max(I_i)$.

The inequalities can be expressed in a more uniform notation. We first note that $I_i \geq \lceil f(\ldots) \rceil$ if and only if $I_i \geq f(\ldots)$ since I_i is an integer. Thus in the inequality $I_i \geq L_i^{\alpha}$ we can remove the ceiling in the L_i^{α}. We can then move the I_i term to the same side as the summation and multiply the inequality through by $l_{i,i}^{\alpha}$ to get

$$l_{i,0}^{\alpha} - l_{i,i}^{\alpha} I_i + \sum_{k=1}^{i-1} l_{i,k}^{\alpha} I_k \leq 0, \text{if } l_{i,i}^{\alpha} > 0.$$

The sense of the inequality is reversed if $l_{i,i}^{\alpha} < 0$. We can perform the same manipulations of the upper bound expressions. We can also multiply through by -1 when the test is $\geq$. Again we can perform the same manipulations for the upper bound inequalities. This results in a series of inequalities of the form

$$e_{i,0}^{\alpha} + \sum_{k=1}^{i} e_{i,k}^{\alpha} I_k \leq 0$$

where the $e_{i,k}^{\alpha}$ are compile time constants and $e_{i,0}^{\alpha}$ is a loop nest invariant.

To determine the loop bounds for loop index i after permutation by σ, we first rewrite each inequality containing i, producing a series of inequalities $i \leq f(\ldots)$ and $i \geq f(\ldots)$. Each inequality of the form $i \leq f(\ldots)$ contributes to the upper bound. If there is more than one such expression, then the minimum of the expressions is the upper bound. Likewise, each inequality of the form $i \geq f(\ldots)$ contributes to the lower bound, and the maximum of the right hand sides is taken if there is more than one. Each inequality of the form $i \geq f(j)$ is considered twice. Suppose loop j is placed outside of i, the expression does not need to be changed since the loop bound of i can be a function of the outer index j. As for loop index j, we must substitute i by its minimum or maximum, whichever minimizes f. A similar procedure is applied to the upper loop bounds.

We demonstrate this method for the above example. Substituting the minimum and maximum of i and j into bounds of loop k, and those of i into bounds of loop j, we obtain:

$$\begin{array}{llll}
k \geq 2i + j - 1 & & k \leq j & k \leq n_k \\
k \geq 3 & & k \leq n_j & k \leq n_k \\
\\
j \geq 2i & k \leq j & j \leq n_j & k \geq 2i + j - 1 \\
j \geq 2i & j \geq k & j \leq n_j & j \leq k - 2i - 1 \\
j \geq 2 & j \geq k & j \leq n_j & j \leq k - 2 - 1 = k - 3 \\
\\
i \geq 1 & & i \leq n_i & j \leq k - 2i - 1 \\
i \geq 1 & & i \leq n_i & i \leq (k - j - 1)/2 = i \leq \lfloor (k - j - 1)/2 \rfloor
\end{array}$$

The transformed bounds are the following:

```
for $k$ := 3 to min($n_k, n_j$) do
  for $j$ := $k$ to min($n_j, k - 3$) do
    for $i$ := 1 to min($n_i, \lfloor (k - j - 1)/2 \rfloor$) do
      $S$;
```

The loop bounds produced as a result of permutation again belong to the class discussed in Section 4.1, so that our methods can calculate the loop bounds after further transformation of these loops.

4.4 Determining the Bounds for General 2-D Loop Transformations

The process for determining the bounds after a general 2-D transformation T is similar to that for permutation. First, we produce the set of inequalities relating the original loop indices i and j, and calculate the maxima and minima for the indices. Transformation T maps i and j to a linear combination of i' and j'. We replace all references to i and j by the equivalent linear combinations of i' and j' in the inequalities. The inequalities remain linear. We then apply the same transformation T to the maxima and minima of i and j to produce those for i' and j'. Once these are known, the loops are placed in the desired order and the bounds are calculated, just as in the permutation case.

4.5 Determining the Bounds after Tiling

It has been suggested that strip-mining and interchanging be applied to determine the bounds of a tiled loop. However, it is not straightforward when the loop bounds are not rectangular [18]. A more direct method is as follows. When tiling, we partition the iteration space, whatever the shape of the bounds, as in Figure 2. Each rectangle represents a computation performed by a tile, some tiles may contain little or even no work.

We replace the loop nest to be tiled, $(p_i, \ldots, p_j)$, with $(p'_i, \ldots, p'_j, p_i, \ldots, p_j)$. The lower bound on the p_k loops, $i \leq k \leq j$, is the maximum of the original lower bound and p'_k; similarly, the upper bound is the minimum of the original upper bound and $p'_k + S_k - 1$, where S_k is the size of the tile in the k

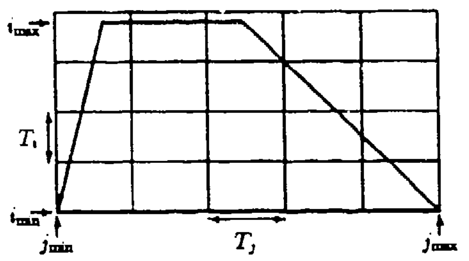

Figure 2: Tiling a trapezoidal loop (2-D)

loop. For loops p'_k, the lower and upper bounds are simply the minimum and maximum values of loop index k. As shown in Figure 2, some of these tiles are empty. The time wasted in determining that the tile is empty should be negligible when compared to the execution of the large number of non-empty tiles in the loop. The p'_k loops step by S_k.

Applying these methods to the permuted example loop nest, we can tile to get the following: (Note that k', j' and i' can be permuted at will.)

```
for $k' := 3$ to $\min(n_j, n_k)$ by $T_k$ do
for $j' := 2$ to $n_j$ by $T_j$ do
for $i' := 1$ to $n_i$ by $T_i$ do
  for $k := \max(3, k')$ to $\min(n_k, n_j, k' + T_k - 1)$ do
  for $j := \max(k, j')$ to $\min(n_j, k - 3, j' + T_j - 1)$ do
  for $i := \max(1, i')$ to $\min(n_i, \lfloor (k - j - 1)/2 \rfloor, i' + T_i - 1)$ do
    $S$;
```

After tiling, the loops *within* the tiles are again in the form we need to perform further permutation, skewing and so on. This property is very useful for tiling for coarse-grain parallelism and then skewing and permuting to create fine-grain DOALL parallelism. The loops *controlling* the tile have a step chosen by the compiler and therefore known at compile time. However, it may not be possible to normalize the loop in such a way that those bounds will still be in the class we need to perform further permutation with loops controlling these tile loops.

5 Conclusions

We have developed a theory that unifies various previously proposed loop transformations, and enables the application of compound transformations. This theory is general enough to encompass both parallelizable and non-parallelizable loops. Previous approaches focus on either specific elementary transformations on general loop nest representation, or general linear transformations on a subclass of loops, namely, the set of loops whose dependences can be represented as a set of distance vectors. This uniform notation is necessary to allow reasoning about the space of all transformations to reduce the search for the optimal transformation.

We have applied this theory to the problem of maximizing the degree of parallelism in loop nests. This paper proposes a practical approach to maximize the degree of parallelism for various different machine architectures via general linear transformations. There are many different possible sequences of linear transformations that can be applied and the algorithm to find the optimal can potentially be expensive. We reduce the problem of maximizing parallelism for different architectures to finding a series of coarsest fully permutable loop nests. By showing that it is easy to transform the loops in this canonical form to suit different architectures, we unify all these different parallelization problems into one. This problem formulation reduces the general parallelization problem into the problem of finding the outermost, largest, fully permutable nest, thus significantly reducing the search space.

We have also applied this theory to the problem of finding loop bounds after transformation. By considering the loop nest as a whole, the general algorithm for determining loop bounds is simplified. For example, in the case of tiling there is no need to have different versions of the transformation for constant loop bounds and for "triangular loops" — the bounds can be determined in a uniform manner.

References

[1] R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. Technical Report Rice COMP TR86-42, Rice University, Nov 1986.

[2] R. Allen and K. Kennedy. Automatic translation of fortran programs to vector form. Technical Report Rice COMP TR84-9, Rice University, Jul 1984.

[3] U. Banerjee. Data dependence in ordinary programs. Technical Report 76-837, University of Illinios at Urbana-Champaign, Nov 1976.

[4] U. Banerjee. A theory of loop permutations. In *2nd Workshop on Languages and Compilers for Parallel Computing*, Aug 1989.

[5] R. Cytron. *Compile-time Scheduling and Optimization for Asynchronous Machines*. PhD thesis, University of Illinois at Urbana-Champaign, 1984.

[6] J.-M. Delosme and I. C. F. Ipsen. Efficient systolic arrays for the solution of toeplitz systems: an illustration of a methodology for the construction of systolic architectures in vlsi. Technical Report 370, Yale University, 1985.

[7] J. A. B. Fortes and D. I. Moldovan. Parallelism detection and transformation techniques useful for vlsi algorithms. *Journal of Parallel and Distributed Computing*, 2:277–301, 1985.

[8] Dennis G., William J., and Kyle G. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5:587–616, 1988.

[9] K Gallivan, W. Jalby, U. Meier, and A. Sameh. The impact of hierarchical memory systems on linear algebra algorithm design. Technical report, University of Illinios, 1987.

[10] F. Irigoin and R. Triolet. Computing dependence direction vectors and dependence cones. Technical Report E94, Centre D'Automatique et Informatique, 1988.

[11] F. Irigoin and R. Triolet. Supernode partitioning. In *Proc. 15th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, January 1988.

[12] M. Lam. Software pipelining: An effective scheduling technique for vliw machines. In *Proc. ACM SIGPLAN 88 Conference on Programming Language Design and Implementation*, pages 318–328, June 1988.

[13] A. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Rice University, May 1989.

[14] P. Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proc. 11th Annual International Symposium on Computer Architecture*, June 1984.

[15] R. Schreiber and J. Dongarra. Automatic blocking of nested loops. 1990.

[16] M. E. Wolf and M. S. Lam. Maximizing parallelism via loop transformations. Technical report, Stanford University, 1990.

[17] M. J. Wolfe. More iteration space tiling. In *Supercomputing '89*, Nov 1989.

[18] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, 1989.